Dissertation

Götz Lindenmaier

Cacheoptimierung für Vererbungshierarchien

Cacheoptimierung für Vererbungshierarchien

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Götz Lindenmaier

aus Göppingen

Tag der mündlichen Prüfung:	23.04.2007
Erster Gutachter:	Prof. em. Dr. Dr. h. c. Gerhard Goos
Zweiter Gutachter:	Prof. Dr. Uwe Aßmann

http://digbib.ubka.uni-karlsruhe.de/volltexte/1000007842

Danksagung

Ich möchte mich bei allen bedanken, die zum Entstehen und Gelingen dieser Dissertation beigetragen haben.

An erster Stelle möchte ich mich bei meinem Doktorvater Prof. em. Dr. Dr. h. c. Gerhard Goos bedanken. Er hat mich immer, auch nach seiner Emeritierung, voll unterstützt. Er war mir stets ein guter Lehrer, hat mich zum wissenschaftlichen Arbeiten angeleitet und mir wissenschaftliches Schreiben vermittelt. An seinem Institut habe ich hervorragende Arbeitsbedingungen vorgefunden. Er hat sich immer für einen guten Zusammenhalt in seiner Gruppe eingesetzt, auf seinen legendären Sommerfesten habe ich nicht zuletzt den Umgang mit dem Sextanten gelernt.

Herrn Prof. Dr. Uwe Aßmann möchte ich für die Übernahme des Koreferats und viele wertvolle Diskussionen danken. Er hat mich vorallem darin bestärkt, nach meinem Diplom eine Promotion anzustreben.

Meiner Frau Jana und meinen Kindern Torben, Lovis und Nils möchte ich für Ihre Unterstützung danken. Jana hat mir den nötigen Freiraum verschafft, und die Kinder mussten an Wochenenden oft auf ihren Papa verzichten.

Meine Kollegen am Institut waren mir eine große Hilfe. Durch ihre Arbeit an Firm haben Martin Trapp, Rainer Neumann, Boris Boesler, Florian Liekweg, Michael Beck und Sebastian Hack entscheidend zu dieser Arbeit beigetragen. Welf Löwe, Martin Spott, Andreas Haeberle, Sabine Glesner, Wolf Zimmermann, Rubino Geiß, Thilo Gaul, Daniela Genius, Jörn Eisenbiegler, Dirk Heuzeroth, Andreas Ludwig, Markus Noga, Elke Pulvermüller, Uwe Wagner und Mamdouh Abu-Sakran, sowie den Kollegen vom FZI, Oliver Ciupke, Holger Bär, Markus Bauer, Thomas Genssler, Helmut Melcher, Benedikt Schulz, Olaf Seng, Adrian Trifu, Volker Kuttruff, Mircea Trifu und Christoph Andressens, möchte ich für Ihre Unterstützung und die gute Stimmung in der Gruppe danken.

Meinen Studenten Rubino Geiß, Olaf Kleine, Sonja Pieper, Christian Schäfer, Hubert Schmid, Matthias Heil, Till Riedel, Andreas Schösser und Beyhan Veliev möchte ich für die Mitarbeit an meiner Forschung danken. Insbesondere Hubert, Matthias, Till und Andreas haben viel zu Firm beigetragen.

Meinen Zimmerkollegen Dirk, Flo und Michael möchte ich danken, dass sie mich so tapfer ausgehalten haben. Annette, Oliver, Jörn und Daniela möchte ich danken, dass Sie mich zum Klettern gebracht haben, und Hagen

Steeger für das ausgiebige Training bis zum 7. Grad. Dadurch hatte ich den nötigen Ausgleich um Stunden über der Cacheoptimierung brüten zu können, ohne mehr als die rechte Hand zu bewegen. Zoltan möchte ich vielmals für seine Motivation danken, weiterzumachen, wenn ich mir mal wieder wünschte, Pförtner bei den Stadtwerken zu sein. Meinen vielen noch nicht genannten Freunden, insbesondere aber Jutta, Iris, Lars und Gerold, möchte ich für die Abwechslung und Entspannung an gemeinsamen Tagen danken.

Schlussendlich möchte ich mich bei der EU und der DFG bedanken, die durch Förderung der Projekte JOSES und ACODA sowie AJACS und CATE diese Arbeit finanziert haben.

Karlsruhe, März 2008
Götz Lindenmaier

Abstract

This thesis introduces an automatic compiler optimization that improves the cache performance of object oriented programs. A static analysis identifies potential hot spots in a program. It combines this information with an analysis of the type structure to select data types from which large data structures are constructed. These are subject to structure splitting and tzpe clustering. We compare the analysis results to profiling data. With both datasets we achieve up to 30% of speed up.

Keywords

cache optimization, object-oriented language, structure splitting, compiler optimization, data structure

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung

Die Leistung von Prozessoren wächst schneller als die Leistung von Speichermedien. Durch diese steigende Disparität kann die Leistung der Prozessoren nicht voll ausgeschöpft werden. Sie müssen auf die weitaus langsameren Speicherzugriffe warten.

Abhilfe schaffen kleine, schnelle Pufferspeicher. Wird ein Datum innerhalb kurzer Zeit mehrmals verwendet, muss es nur einmal in den Pufferspeicher geladen werden. Danach kann der Prozessor von der schnellen Zugriffszeit des Pufferspeichers profitieren. Da die Pufferspeicher in kleinen Blöcken verwaltet werden, kann der Prozessor auch bei Zugriffen auf im Hauptspeicher benachbarte Daten die geringe Latenz des Pufferspeichers ausnutzen. Die zeitliche oder räumliche Wiederverwendung der Daten manifestiert sich in Lokalität.

Wird zwischen zwei Zugriffen auf das gleiche Datum, oder auch benachbarte Daten, eine große Menge weiterer Daten angefasst, werden die bereits geladenen Daten aus dem Pufferspeicher verdrängt. Aufgrund der in der Elektronik festgelegten Abbildung des Hauptspeichers auf den Pufferspeicher kann dies auch weitaus früher geschehen. Es ergibt sich keine Lokalität der Daten im Pufferspeicher.

Hier setzen Programmoptimierungen an. Diese formen Algorithmen so um, dass diese möglichst lange auf einer kleinen Teilmenge der Daten arbeiten, um diese im Pufferspeicher zu halten. Oder sie beeinflussen die Anordnung der Daten im Hauptspeicher, sodass die Abbildung auf Pufferspeicher

günstiger ausfällt. Diese Optimierungen können sowohl vom Programmierer als auch automatisch durch den Übersetzer angebracht werden. Diese Arbeit beschäftigt sich mit Übersetzeroptimierungen der Pufferleistung.

Die meisten in der Literatur vorgeschlagenen Optimierungen der Pufferleistung zielen auf numerische Anwendungen. Diese zeichnen sich durch leicht analysierbare Schleifen aus, die auf Reihungen operieren. Für Zeigeranwendungen sind weitaus weniger Optimierungen bekannt.

Algorithmen in Zeigeranwendungen sind weitaus schwerer analytisch zu erfassen als Algorithmen in numerischen Anwendungen. Diese sind häufig von geschachtelten Zählschleifen dominiert. Iterationen in Zeigeranwendungen sind oftmals durch Rekursionen oder mit Iteratoren implementiert und somit nicht direkt an einem Sprachkonstrukt zu erkennen. Zum Beispiel sind die Voraussetzungen für das korrekte Vertauschen geschachtelter Schleifen zur Pufferoptimierung von Reihungen mit statischen Analysen leichter zu erkennen, als Voraussetzungen um eine Tiefensuche durch eine Breitensuche zu ersetzen.

Bei der Anordnung von Datenstrukturen verhält es sich umgekehrt. Numerische Algorithmen arbeiten in der Regel auf Reihungen, die große Speicherbereiche belegen. Die Anordnung der Daten innerhalb der Reihungen lässt sich nur in beschränktem Maße verändern. Zeigeranwendungen operieren hingegen auf vielen, relativ kleinen Verbunden. Deren relative Anordnung ist nicht vorgegeben. Wird ein Speicherbereiniger verwendet, kann sich die Anordnung während der Lebenszeit eines Verbunds ändern.

Aufgrund der schlechten analytischen Fassbarkeit von Zeigeranwendungen sind viele bekannte Optimierungen für Datenstrukturen nur teilweise automatisiert. Eine Optimierung durch den Programmierer oder separate Werkzeuge zur Pufferoptimierung automatisieren die Optimierung in unseren Augen nicht ausreichend. Optimierungen, die speziellen Einsatz des Programmierers benötigen, müssen durch extreme Pufferprobleme begründet sein. Vollständig automatisierte Optimierungen dagegen benötigen keine Interaktion des Programmierers. Sie sind leichter anzuwenden und können damit häufiger praktische Pufferprobleme vermeiden.

1.2 Zielsetzung der Arbeit

Das **Ziel dieser Arbeit** *ist, eine Optimierungsstrategie für Zeigeranwendungen aufzuzeigen, die in herkömmlichen Übersetzern eingesetzt werden kann.*

Dabei soll insbesondere auf die Probleme der Optimierung objektorientierter Programme eingegangen werden die sich unter anderem aus Vererbungshierarchien ergeben.

Die Optimierung soll nicht nur die Kosten eines Fehlzugriffs auf den Pufferspeicher verringern, wie dies zum Beispiel durch Vorladen erreicht werden kann. Sie soll insbesondere die Anzahl der Fehlzugriffe verringern. Dies verringert ebenfalls die durchschnittlichen Kosten von Speicherzugriffen, entlastet aber zusätzlich die für Speicherzugriffe verwendete Hardware. Weiter werden Zeigeranwendungen häufig von kleinen Methoden und häufigen Entscheidungen [LW92] dominiert, wodurch viele Sprünge und kleine Grundblöcke entstehen. Dies erschwert Vorladebefehle einzufügen oder Ladebefehle weit vor der Verwendung des geladenen Datums im Programm zu platzieren. Die Latenz der Ladebefehle kann also schlechter kaschiert werden als bei rechenintensiven numerischen Anwendungen.

Bei der Optimierung von Zeigeranwendungen stellen sich verschiedene Herausforderungen. In großen Programmen wird eine Vielzahl verschiedener Datenstrukturen definiert. Viele dieser Datenstrukturen stellen kein Problem für die Pufferleistung dar, da sie selten verwendet werden oder da die Verwendung nur einen geringen Umfang hat. Eine Pufferoptimierung muss also die pufferrelevanten Datenstrukturen erkennen können. Rechenintensive Algorithmen in Programmen stellen ein geringeres Pufferproblem dar als datenintensive Algorithmen. Diese fassen in kurzer Zeit viele Daten an, sodass das Fassungsvermögen des Pufferspeichers überschritten wird. Eine Pufferoptimierung muss die problematischen datenintensiven Algorithmen in einem Programm identifizieren.

Anhand dieser Informationen muss sich die Pufferoptimierung für eine Optimierungsstrategie entscheiden, die die Performanz der pufferkritischen Programmstellen verbessert ohne an anderer Stelle negative Auswirkungen zu haben. Dabei muss natürlich sicher gestellt werden, dass die Transformationen des Programms die Semantik des Programms erhalten. Dabei stellen Programme mit Vererbung ein Problem dar. Eine Instanz kann abhängig von ihrer dynamischen Interpretation als der eine oder andere Typ andere Anforderungen an den Pufferspeicher stellen. Die Optimierung des Layout eines Verbunds wird durch Forderungen nach Typkonvertierbarkeit innerhalb der Vererbungshierarchie erschwert. Die Transformation eines einzelnen Typs kann eine Transformation des Obertyps bedingen, die sich auf alle anderen Untertypen dieses Obertyps fortpflanzt. Ist die Transformation als Optimierung des ersten Typs gedacht, kann diese negative Auswirkungen auf die

anderen transformierten Typen haben. Hier muss eine Pufferoptimierung geeignete Lösungen finden.

Diese Arbeit stellt eine statische Analyse vor, die pufferkritische Verbunde erkennt, und definiert eine darauf aufbauende Optimierung der Pufferleistung. Als Grundlage der Optimierungen definieren wir eine Zwischensprache, die für die Analyse und Optimierung wichtige Programmdetails effizient darstellt. Die Zwischensprache verbindet insbesondere Hochsprachtypisierung der Programmkonstrukte und Datenstrukturen mit einer prozessornahen Darstellung des Programmcodes und der Datenanordnungen. Dadurch kann die Analyse Informationen aus der Typisierung des Programms, insbesondere bei der Übersetzung vollständig getypter Hochsprachen, ableiten. Die Optimierung kann wiederum gezielt das Programm ändern und genau die Eigenschaften erhalten, die von der Hochsprache gefordert sind und im Programm auch benötigt werden. Die prozessornahe Darstellung ermöglicht andererseits, die Pufferoptimierung nach verschiedenen anderen Optimierungen auszuführen, die die Effizienz der Pufferoptimierung steigern.

Wir stellen erstmals eine statische Analyse vor, die mittels einer Heuristik pufferkritische Datenstrukturen erkennt. Die Literatur kennt zur Optimierung von Zeigeranwendungen durch Reduktion von Fehlzugriffen ausschließlich Algorithmen, die auf Profildaten basieren. Wir validieren unsere Heuristik im Vergleich mit Profildaten.

Zur Optimierung der Pufferleistung stellen wir Typanhäufen vor, eine Technik, die grobgranularer ansetzt als das aus der Literatur bekannte Anhäufen, aber deutlich weniger Zusatzkosten verursacht. Typanhäufen platziert häufig gemeinsam verwendete Verbunde nah beieinander und separiert sie von selten genutzten Verbunden. Das Ziel ist, im Mittel den Pufferspeicher besser auszunutzen. Um den Effekt von Typanhäufen zu verstärken, setzen wir Verbundteilen ein. Verbundteilen bewirkt, dass selten genutzte Felder in Verbunden nicht von Typanhäufen betroffen sind, sodass die häufig verwendeten Felder durch Typanhäufen stärker profitieren können. Wir erweitern Verbundteilen für Vererbungshierarchien und formulieren es als echte Übersetzeroptimierung. Eine zweite Heuristik entscheidet wie pufferkritische Verbunde mit diesen Optimierungen transformiert werden. Wir vergleichen die Optimierungen auf Basis unserer Analyse mit den gleichen Optimierungen auf Basis von Profilinformationen.

Die Arbeit gliedert sich wie folgt: In Kapitel 2 definieren wir die in dieser Arbeit verwendeten Begriffe und erläutern das von uns adressierte Problem im Detail. Kapitel 3 klassifiziert bekannte Algorithmen und zeigt, wie sie in

einem Übersetzer Anwendung finden. Kapitel 4 definiert die Zwischensprache Firm die wir gezielt für die in dieser Arbeit entwickelten Optimierungen angepasst haben. In Kapitel 5 beschreiben wir eine statische Analyse, die pufferkritische Datenstrukturen und Algorithmen auf Firm identifiziert. Kapitel 6 stellt die darauf aufbauenden Optimierungen dar. In Kapitel 7 vergleichen wir die Analyse mit Profildaten und diskutieren die durch die Optimierung erreichten Effekte. Kapitel 8 fasst zusammen und gibt einen Ausblick.

Kapitel 2

Grundlagen und Problemanalyse

Dieses Kapitel erläutert die Organisation des Speichers in modernen Prozessoren und führt die daraus entstehenden Probleme ein. Es definiert die Begriffe, die diese Probleme charakterisieren.

Wir betrachten im Folgenden Einprozessorsysteme, da wir uns in dieser Arbeit nicht mit Parallelisierung oder Effekten zwischen verschiedenen Programmen auf Mehrprozessorsystemen auseinandersetzen.

2.1 Die Speicherhierarchie in modernen Rechnern

Moderne Prozessoren verwenden *Hauptspeicher* mit mehreren Gigabyte Kapazität. Die Adressierungslogik solch großer Speicher ist langsamer als die kleiner Speicher. Kleine Speicher können in einer teureren Technologie produziert werden. Sie können auf dem Prozessorchip platziert werden. Dies ist mit großen Speichern aufgrund ihrer schieren Größe, und da diese in einer grundlegend anderen, billigeren Technologie gefertigt sind, nicht möglich. Dadurch sind kleine Speicher deutlich schneller, aber teurer als große.

Wir messen die Geschwindigkeit eines Speichers mit der Latenzzeit. Die Latenzzeit eines Speichers ist die Anzahl Prozessortakte zwischen dem Beginn der Ausführung eines Ladebefehls und dem Zeitpunkt, zu dem das Datum zur Verarbeitung zur Verfügung steht. Die Dauer eines Speicherbefehles ist für die Laufzeit weniger relevant, da er keine der für die Programmausführung

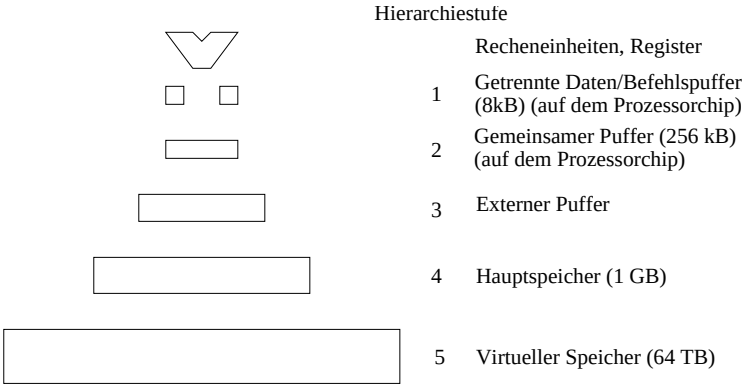

Abbildung 2.1: Typische Speicherhierarchie

notwendigen Register definiert. Sie erhöhen jedoch die Last auf dem Speicherbus und vergrößern dadurch indirekt die Latenz von Ladebefehlen.

Um schnelle und große Speicher zu kombinieren, verwenden moderne Rechner eine mehrstufige Hierarchie von Speichern. Die Hierarchiestufe, auf die der Prozessor direkt zugreift, nennen wir die höchste oder erste Stufe.

Die verschiedenen Speicher unterscheiden sich in der Art ihrer Adressierung. Das Programmiermodell des Prozessors zusammen mit dem Betriebssystem beschränkt die Größe des adressierbaren Speichers. Dadurch ist der *virtuelle Speicher* eines Systems definiert. Befehle des Programms können den virtuellen Speicher direkt adressieren. Wir fassen den virtuellen Speicher als unterste Speicherstufe auf.

Das Betriebssystem bildet Adressen des virtuellen Speichers auf den Hauptspeicher ab. Die physikalischen Adressen im Hauptspeicher werden vom Betriebssystem, in der Regel mit Hardwareunterstützung, aus den virtuellen Adressen gebildet. So genannte *Pufferspeicher* werden durch Hardware adressiert. Diese Hardware bildet reale Hauptspeicheradressen auf reale Adressen im Pufferspeicher ab.

Jedes Datum eines Programms ist einer virtuellen Adresse oder einem Register zugeordnet. Bevor ein Datum aus einem Register in den Speicher geschrieben werden kann, muss es einer virtuellen Adresse zugeordnet werden. Wir betrachten im Folgenden Daten in Registern nicht mehr.

Definition 2.1.1 (Zelle) *Eine Zelle ist die kleinste adressierbare Dateneinheit mit einer Adresse im virtuellen Speicher.*

Zellen werden bei einem Zugriff in den Speicherhierarchiestufen oberhalb des virtuellen Speichers alloziert. Dazu wird die virtuelle Adresse auf reale Adressen der verschiedenen Speicherhierarchiestufen abgebildet. Ist für eine Zelle in einem Speicher der Hierarchiestufe n Platz alloziert, sagen wir, die Zelle *ist* in dem Speicher der Stufe n.

Axiom 2.1.1 *Ist eine Zelle im Speicher der Stufe n, so ist sie auch in allen Speichern der Stufen n' mit $n' > n$.*

2.1.1 Adressabbildung

In der Regel sind physikalische Speicher kleiner als der virtuelle Speicher, insbesondere Pufferspeicher. Daher können die Abbildungen der virtuellen Adressen auf reale nicht bijektiv sein. Mehrere Zellen werden also auf die gleiche reale Adresse abgebildet. Werden Zellen, die auf die gleiche reale Adresse abgebildet sind, gleichzeitig verwendet, entstehen Konflikte. Um solche Konflikte zu vermeiden, ist die Abbildung oft nicht nur von der virtuellen Adresse, sondern auch vom Zustand des Programms bzw. der Belegung des Speichers abhängig.

Da die Abbildung auf reale Adressen des Hauptspeichers vom Betriebssystem in Software ausgeführt wird, können hier beliebige Strategien implementiert werden.

Die Abbildung auf Adressen eines Pufferspeichers ist die Verknüpfung der Abbildung der virtuellen Adresse auf eine Adresse des Hauptspeichers, und einer in Hardware implementierten Abbildung der Hauptspeicheradresse auf die Pufferadresse[1]. Im folgenden diskutieren wir die Teilabbildung Hauptspeicheradresse auf Pufferadresse.

Es gibt Pufferspeicher, bei denen die Adressabbildung nicht von der Belegung des Speichers abhängt.

Definition 2.1.2 (direkt abbildender Pufferspeicher) *Bei einem direkt abbildenden Pufferspeicher ist die Adressabbildung nur von der Hauptspeicheradresse abhängig.*

In der Regel sehen Pufferspeicher jedoch mehrere alternative Pufferadressen für eine Hauptspeicheradresse vor.

[1]Es gibt auch Puffer, die mit virtuellen Adressen adressiert werden. Hier wird die Abbildung auf die Hauptspeicheradresse nicht benötigt.

Definition 2.1.3 (assoziativer Pufferspeicher) *Ein n-fach assoziativer Pufferspeicher kann eine Hauptspeicheradresse auf n verschiedene Pufferadressen abbilden.*

Eine in Hardware implementierte Strategie wählt eine der n möglichen Adressen aus. In der Regel implementieren Puffer hier eine ‚gerade am wenigsten benutzt' (least recently used) Strategie. Damit ist die Abbildung vom Zustand des Programms abhängig.

Ein Speicher ist voll assoziativ, wenn jede Zelle an jeder Position alloziert werden kann.

2.1.2 Schreiben und Verdrängen

Schreibt ein Programm ein Datum in eine Zelle, muss diese zunächst im obersten Speicher alloziert werden. Dann wird das Datum in die Zelle in diesem Speicher geschrieben. Die Zelle ist dann nach Axiom 2.1.1 auch in allen darunter liegenden Speichern alloziert. Das neue Datum muss nicht notwendiger Weise in alle darunter liegenden Speicher kopiert werden, da bei einem weiteren Zugriff das Datum in der Zelle im obersten Speicher gefunden wird. Solange die Zelle im obersten Speicher ist, finden keine Zugriffe auf die Zelle in Speichern darunter statt.

Wird eine Zelle an einer Adresse alloziert, auf die bereits eine andere Zelle alloziert wurde, muss diese Zelle aus dem Speicher verdrängt werden. Wird eine Zelle aus Speicher n verdrängt, wird sie nicht unbedingt aus Speicher $n+1$ verdrängt, jedoch auch aus allen höheren Speichern $m, m < n$. Spätestens beim Verdrängen wird ein geschriebenes Datum in den darunter liegenden Speicher $n + 1$ kopiert.

In dieser Arbeit betrachten wir keine Veränderungen des Hauptspeichers durch andere Systemkomponenten wie durch Direktspeicherzugriff (DMA) oder in Mehrprozessorsystemen.

2.1.3 Zeilen und Seiten

Speicher werden nicht in einzelnen Zellen verwaltet. Es werden immer Blöcke mehrerer Zellen mit aufeinander folgenden Adressen gemeinsam in einen Speicher geladen. Dies vereinfacht die Adressverwaltung der Zellen. Es muss nur die Abbildung der realen Anfangsadresse eines Blocks auf die virtuelle Anfangsadresse gemerkt werden. Weiter reduziert dies den Aufwand des

Größe	1. D/B Puffer	2. Puffer	3. Puffer	Haupt-speicher
Speichergröße	16 kB	256 kB	min 1536 kB	mehrere GB
Seiten/Zeilengröße	64 B	128 B	128 B	8 kB
Assoziativität	4-fach	8-fach	128-fach	vollassoziativ
Latenzzeit	1 Takt	5+ Takte	12+ Takte	$\gg$12 Takte

Tabelle 2.1: Beispielhafte Speicherhierarchie eines Itanium 2 Systems [LDMM02].

Ladens. Laden n aufeinander folgender Zellen ist effizienter als n beliebige Zellen zu laden. Z.B. muss nur eine Ladeanforderung an den Speicher gesendet werden um mehrere Zellen zu laden.

Schließlich hofft man, durch das Laden ganzer Blöcke auch Zellen zu laden, die in naher Zukunft, bevor der Block verdrängt wird, verwendet werden. Diese werden dann bereits im Speicher vorgefunden.

Die Zellenblöcke im Hauptspeicher nennt man *Seiten*. Sie enthalten zum Beispiel 8 KB Daten. Die Zellenblöcke in Pufferspeichern nennt man *Zeilen*. Sie sind weitaus kleiner als Seiten und enthalten zum Beispiel 32 B Daten. Ist ein Speicher in Zeilen/Seiten organisiert, spricht man bereits von einem voll assoziativen Speicher wenn jede Zeile/Seite an beliebiger Stelle alloziert werden kann.

Es gibt Experimente mit Pufferspeichern, die von den hier beschriebenen Speichern abweichen. Wir beschäftigen uns jedoch mit Optimierungen für heutige Prozessoren die vorrangig die beschriebenen Pufferspeicher einsetzen.

2.2 Speicherzugriffe

Bei einer Programmausführung lädt die CPU Befehle und Daten, das heißt die Inhalte von Zellen.

Definition 2.2.1 (Speicherzugriff) *Ein Speicherzugriff ist das Laden oder Schreiben des Inhalts einer Zelle während der Programmausführung.*

Speicherzugriffe, kurz Zugriffe, werden im Laufe eines Programms ausgeführt. Eine Programmstelle, die Speicherzugriffe initiiert, nennen wir Speicherreferenzstelle oder kurz Referenzstelle.

Definition 2.2.2 (Referenzstelle) *Eine Referenzstelle ist eine Programmstelle, die Speicherzugriffe initiiert.*

Eine Referenzstelle kann während eines Programmlaufs mehrere Speicherzugriffe auf verschiedene Zellen initiieren. Jeder Befehl eines Programms ist eine Referenzstelle, da er das Laden des danach auszuführenden Befehls initiiert. Interessanter sind Referenzstellen, die Daten laden. Im Folgenden betrachten wir nur solche Referenzstellen.

Wir unterscheiden drei Arten von Referenzstellen. Referenzstellen die auf statisch bekannte Adressen zugreifen, Referenzstellen die auf statisch vorhersagbare Adressen zugreifen, und solche die auf unvorhersagbare Adressen zugreifen. Die beiden ersten nennen wir kurz *statische Referenzstellen*, letztere *dynamische Referenzstellen*. Statische Referenzstellen initiieren *statische Zugriffe*, dynamische Referenzstellen *dynamische Zugriffe*.

2.3 Wiederverwendung und Lokalität

Die Begriffe Wiederverwendung und Lokalität sind Eigenschaften eines Programms, die beschreiben, ob das Programm von einem Pufferspeicher profitieren könnte oder bereits profitiert. Sie bewerten zunächst einen einzelnen Zugriff, und können zu einer Programmeigenschaft summiert werden.

Wird bei der Ausführung eines Programms nur einmal auf eine Zelle zugegriffen, muss diese aus dem untersten Speicher geladen werden. Pufferspeicher können den ersten Zugriff nicht beschleunigen. Wird jedoch mehrmals auf die Zelle zugegriffen, kann diese bei dem zweiten und späteren Zugriffen in einem Pufferspeicher sein und die schnellere Zugriffszeit des Puffers ausgenutzt werden.

Definition 2.3.1 (Temporale Wiederverwendung) *Ein Speicherzugriff mit temporaler Wiederverwendung greift auf eine Zelle zu, auf die bereits vorher zugegriffen wurde.*

Da Pufferspeicher in Zeilen organisiert sind, kann eine Zelle bereits in einem Puffer gewesen sein, bevor jemals auf sie zugegriffen wurde. Sie wurde in den Puffer geladen, als auf eine andere Zelle in der Zeile zugegriffen wurde, und später eventuell wieder verdrängt.

Definition 2.3.2 (Wiederverwendung in Speicher n) *Ein Speicherzugriff mit Wiederverwendung in Speicher n ist ein Zugriff auf eine Zelle, die in einer Zeile ist, auf die bereits vorher zugegriffen wurde.*

Die Zeilengröße und die Ausrichtung der Zeile bestimmen, welche weiteren Zellen mit einer zugegriffenen Zelle geladen werden. Daher ist Wiederverwendung nur in Abhängigkeit eines spezifischen Speichers definiert.

Für Pufferoptimierungen ist es interessant zu wissen, warum eine Zelle zuletzt im Puffer war.

Definition 2.3.3 (Direkte temporale Wiederverwendung in Speicher n) *Ein Speicherzugriff hat direkte temporale Wiederverwendung in Speicher n wenn der* letzte Zugriff *auf diese Zeile auf* genau diese *Zelle zugriff.*

Definition 2.3.4 (Direkte räumliche Wiederverwendung in Speicher n) *Ein Speicherzugriff hat direkte räumliche Wiederverwendung in Speicher n wenn der* letzte Zugriff *auf diese Zeile* nicht *auf diese Zelle zugriff.*

Direkte temporale Wiederverwendung hängt vom Speicher ab, da die Zeilengröße bestimmt, wann eine Zeile zuletzt im Pufferspeicher war. Wird auf eine Zeile mehrmals zugegriffen, kann ein Zugriff auf eine Zelle temporale und räumliche Wiederverwendung haben. Durch die Einschränkung auf den letzten Zugriff auf die Zeile kann die Art der Wiederverwendung eindeutig angegeben werden.

Aufbauend auf dem Begriff der Wiederverwendung können wir nun Lokalität definieren. Die Lokalität ist ein Maß, das die tatsächliche Pufferleistung eines Programms beschreibt. Ein Zugriff ist lokal, wenn er Wiederverwendung hat und die Zeile seit dem letzten Zugriff nicht aus dem Puffer verdrängt wurde.

Definition 2.3.5 (Lokalität in Speicher n) *Ein Zugriff mit Lokalität in Speicher n ist ein Zugriff mit Wiederverwendung in Speicher n. Die zugegriffene Zeile wurde seit dem letzten Zugriff nicht aus Speicher n verdrängt.*

Definition 2.3.6 (Räumliche Lokalität in Speicher n) *Ein Zugriff mit räumlicher Lokalität in Speicher n ist ein Zugriff mit Lokalität in Speicher n mit direkter räumlicher Wiederverwendung.*

Definition 2.3.7 (Temporale Lokalität in Speicher n) *Ein Zugriff mit temporaler Lokalität in Speicher n ist ein Zugriff mit Lokalität in Speicher n mit direkter temporaler Wiederverwendung.*

Die Begriffe Wiederverwendung (reuse) und Lokalität (locality) werden in der Literatur gegensätzlich definiert.

Wolfe und Lam [WL91, S.34] definieren Wiederverwendung als eine inhärente Eigenschaft einer Schleife: „*A data item is reused if the same data is used in multiple iterations in a loop nest. Thus reuse is a measure that is inherent in the computation...*“. Lokalität definieren sie als durch den Puffer ausgenutzte Wiederverwendung: „*utilized reuse*“. Verschiedene andere Arbeiten schließen sich dieser Definition an ([Car96]).

Temam und McKinley [MT99, S.5] vertauschen die Bedeutung dieser beiden Begriffe. Sie sagen „*References with locality thus have the potential for reuse in the cache. Reuse is simply a hit in the cache that achieves locality.*“ Das heißt, sie definieren Lokalität als inheränte Programmeigenschaft, wohingegen Wiederverwendung die tatsächliche Mehrfachverwendung der Daten im Puffer beschreibt. Die Definitionen räumlicher und zeitlicher Lokalität unterscheiden sich ebenfalls. [WL91] definieren zeitliche Lokalität als einen Sonderfall räumlicher Lokalität, [MT99] definieren sie als exklusive Eigenschaft.

Unsere Definitionen (2.3.1 – 2.3.7) sind präziser als die Definitionen aus der Literatur, halten sich aber in den Begriffen an die ältere Variante von [WL91]. In der Unterscheidung räumlich von temporal folgen wir der genaueren Definition von [MT99].

Ob sich Wiederverwendung als Lokalität ausnutzen lässt, hängt im Allgemeinen davon ab, wie viele Daten zugegriffen werden seitdem eine Zelle im Pufferspeicher alloziert wurde.

Definition 2.3.8 (Zugriffsabstand) *Der Zugriffsabstand von zwei Zugriffen auf die gleiche Zelle oder die gleiche Zeile ist die Anzahl Zugriffe die zwischen diesen beiden Zugriffen ausgeführt werden.*

Haben zwei Zellen einen geringen Zugriffsabstand, kann ein Pufferspeicher die Wiederverwendung der Zellen als Lokalität ausnutzen. Die Wahrscheinlichkeit, dass die Zellen vor einer weiteren Benutzung verdrängt werden, ist gering.

2.4 Probleme mit Speicherzugriffen

Speicherzugriffe auf einen Speicher n sind effizient, wenn der Wert in diesem Speicher vorgefunden wird.

Definition 2.4.1 (Treffer) *Ein Speicherzugriff auf eine Zelle im Speicher der Hierarchiestufe n ist ein Treffer, wenn die Zelle in diesem Speicher ist.*

Korollar 2.4.1 *Ein Treffer ist ein Zugriff mit Lokalität.*

Ein Speicherzugriff auf einen Speicher n ist problematisch, wenn der Wert dort nicht vorgefunden wird.

Definition 2.4.2 (Fehler) *Ein Speicherzugriff auf eine Zelle im Speicher der Hierarchiestufe n ist ein Fehler, wenn die Zelle* nicht *in diesem Speicher ist.*

Ist ein Zugriff auf Speicher n ein Fehler, ist ein Zugriff auf Speicher $n+1$ notwendig. Daher ist eine Zelle nach einem Zugriff auf diese Zelle in allen unter Speicher n liegenden Speichern.

Die Zugriffe einer Referenzstelle können Treffer und Fehler sein. So kann der erste Zugriff einer statischen Referenzstelle ein Fehler sein, darauf folgende jedoch Treffer. Wird die Zelle von einem anderen Zugriff aus einem Speicher verdrängt, tritt wieder ein Fehler auf.

Wir können Fehler anhand der Eigenschaft des Pufferspeichers, die den Fehler bedingt, weiter unterscheiden.

Definition 2.4.3 (Obligatorischer Fehler) *Ein Zugriff ohne Wiederverwendung ist ein obligatorischer Fehler.*

Obligatorische Fehler treten beim ersten Zugriff auf eine Zeile auf. Um obligatorische Fehler zu vermeiden, muss man die Wiederverwendung eines Programms verbessern.

Die folgenden Fehler können durch eine Optimierung der Lokalität vermieden werden. Die Zugriffe haben also Wiederverwendung aber keine Lokalität. Die begrenzte Größe der Speicher ist eine Ursache für Fehler.

Definition 2.4.4 (Kapazitätsfehler in Speicher n) *Ein Kapazitätsfehler ist ein Fehler mit Wiederverwendung, bei dem seit dem letzten Zugriff auf diese Zeile auf mehr* Zellen *zugegriffen wurde als in Speicher n passen.*

Die Kapazität des Speichers reicht nicht, wenn die Datenmenge auf der das Programm operiert nicht in den Speicher passt. Eine Zeile kann jedoch bereits verdrängt sein, wenn auf eine Datenmenge, die nicht den gesamten Speicher füllt, zugegriffen wurde. Wird auf nur eine Zelle in jeder Zeile zugegriffen, reichen Speichergröße/Zeilengröße Zugriffe, um eine Zeile zu verdrängen.

Definition 2.4.5 (Zeilen-Kapazitätsfehler in Speicher n**)** *Ein Zeilen-Kapazitätsfehler ist ein Fehler mit Wiederverwendung, bei dem seit dem letzten Zugriff auf diese Zeile auf mehr* Zeilen *zugegriffen wurde als in Speicher* n *passen.*

In der Literatur werden Kapazitätsfehler und Zeilen-Kapazitätsfehler nicht genau unterschieden. Bei einem Speicher mit Zeilengröße Eins fallen Kapazitätsfehler und Zeilen-Kapazitätsfehler zusammen.

Die restriktive Adressabbildung der Pufferspeicher ist eine weitere Ursache für Fehler.

Definition 2.4.6 (Konfliktfehler in Speicher n**)** *Ein Konfliktfehler ist ein Fehler mit Wiederverwendung, der kein Kapazitätsfehler ist. Seit dem letzten Zugriff auf diese Zeile wurde auf* $m > a$ *Zeilen zugegriffen, wobei* a *die Assoziativität des Speichers ist.*

Obwohl die Kapazität des Speichers noch nicht ausgenutzt wurde, wurde die Zeile bereits verdrängt. Konfliktfehler treten bei voll assoziativen Speichern, wie dem Hauptspeicher, nicht auf.

In [MT96, MT99] findet sich eine Untersuchung der Gründe mangelhafter Pufferleistung in bekannten Testprogrammen.

2.4.1 Kosten eines Fehlers

Fehler können die Laufzeit eines Programms enorm beeinflussen.

Wir betrachten ein System mit einer Speicherhierarchie wie in Tabelle 2.1, das 5 Befehle parallel ausführen kann. Wir nehmen an, dass jeder 20. ausgeführte Befehl ein Fehler im 1. und 2. Puffer ist, jedoch ein Treffer im 3. Puffer. Der Befehl dauert dann 11 Takte länger als ein Treffer im 1. Puffer.

Das System kann, während es auf die Erledigung des Zugriffs wartet, 11*5 = 55 Befehle ausführen. Hat das Programm ausreichend Parallelität

treten in dieser Zeit trotzdem 2 weitere Fehler auf. Kann das System 2 Fehler gleichzeitig verarbeiten, verlängert sich spätestens beim 3. Fehler die Latenzzeit zusätzlich.

Ein EPIC Prozessor, der Befehle nicht dynamisch umordnen kann, muss bei einem Fehler die Ausführung anhalten. Es gehen 11 Takte bzw. 55 Befehle verloren. Geht man davon aus, dass alle anderen Befehle wie vorgesehen ausgeführt werden, wird die Ausführung nach 4 Takten für 11 Takte angehalten. Damit sind ca. 11/15 bzw. 73% der Ausführungszeit durch Fehler bedingt.

Damit könnten Optimierungen der Pufferleistung bis zu 73% der Ausführzeit von Programmen reduzieren. Dies ist jedoch eine obere Schranke. Obligatorische Fehler können nur sehr beschränkt reduziert werden. Andere Fehler können nur zu einem Teil reduziert werden. Wird jeder fünfte Fehler vermieden, ist noch jeder 25. Befehl ein Fehler. Damit sind 11/16 bzw. 69% der Ausführzeit von Fehlern verschuldet. Die Programmlaufzeit wurde um 4% reduziert.

2.5 Daten in Programmen

Pufferoptimierungen sind nur nötig, wenn Programme große Datenmengen innerhalb geringer Zeit verwenden. Daten sind Zahlen und Zeichenketten, aber auch Adressen anderer Daten. Adressen, die als Daten durch das Programm manipuliert werden, nennen wir auch *Referenzen*.

Ein Programm muss Variablen enthalten, die diese Daten fassen können. Um mit vielen Daten umzugehen, verwendet man Variablen die mehrere einzelne Daten zusammenfassen.

Definition 2.5.1 (Reihung) *Im Speicher aufeinander folgende Daten gleicher Größe, die über eine Basis anhand von Indizes zugegriffen werden, heißen Reihung.*

Die einzelnen Daten in einer Reihung nennen wir *Reihungsfeld*er oder auch einfach *Feld*er. Die Felder werden indirekt durch ihre Anzahl spezifiziert. Dadurch lassen sich leicht große Datenmengen beschreiben. Zugriffe auf Reihungsfelder sind statische Zugriffe. Wir betrachten Zugriffe auf Reihungen auch dann als statisch, wenn die Basis nur dynamisch bekannt ist, da bei Reihungen die Effekte der Felder untereinander eine große Rolle spielen. Diese Effekte lassen sich auch dann statisch analysieren, wenn die Basis nicht bekannt ist.

Ein Programm, das vorrangig auf Reihungen zugreift, nennen wir eine *numerische Anwendung*. Algorithmen in numerische Anwendungen verwenden häufig Schleifenschachteln mit Zählschleifen, deren Rümpfe auf Reihungen operieren. Anhand der Laufvariablen der Schleifen kann man die Reihungszugriffe vorhersagen, die Schleifen enthalten also statisch vorhersagbare Zugriffe. Die Pufferprobleme dieser Algorithmen sind analytisch leicht zu erfassen. Reihungszugriffe können auf Datenabhängigkeiten untersucht werden [Wol89, Ban88].

Reihungen zeichnen sich insbesondere dadurch aus, dass ihre Felder alle den gleichen Datentyp haben. Darin unterscheidet sich ein Verbund.

Definition 2.5.2 (Verbund) *Ein Verbund ist eine Ansammlung verschiedenartiger Daten.*

Wir nennen die einzelnen Daten eines Verbunds *Verbundfelder* oder einfach *Feld*er. Die Felder werden durch Aufzählung spezifiziert. In einem Programm viele Felder aufzuzählen ist nicht sinnvoll. Daher ist die Größe von Verbunden deutlich geringer als die von Reihungen. Dafür verwenden Programme viele gleichartige Verbunde. Hochsprachen beschreiben gleichartige Verbunde durch einen Typ (`struct` in C, `class` in Java). Da es nicht sinnvoll ist, viele Verbunde statisch zu allozieren, werden diese dynamisch alloziert und können frei im Speicher platziert werden. Zugriffe auf Felder sind dynamische Zugriffe. Die relative Position eines Feldes in einem Verbund ist statisch bekannt. Da Verbunde jedoch klein sind, spielt die nur dynamisch bekannte Basis des Verbunds die entscheidende Rolle.

In Hochsprachen können Felder eines Verbunds wiederum Verbunde (oder Reihungen) sein. Um eine korrekte Übersetzung zu ermöglichen, müssen diese zur Übersetzungszeit vollständig bekannt sein. Der Übersetzer kann die inneren Verbunde einsetzen. Daher betrachten wir im Folgenden nur Verbunde bei denen alle Felder atomare Werte enthalten, die in einer Zelle des Speichers abgelegt werden können.

Objektorientierte Sprachen verwenden *Vererbung*, um die Definitionen von Verbundtypen zu vereinfachen. Dabei werden gemeinsame Eigenschaften unterschiedlicher Verbunde ausfaktorisiert und in einem gemeinsamen Teiltyp beschrieben. Diesen gemeinsamen Teiltyp nennt man *Obertyp*. Bei der Definition eines Verbundtypen kann man einen oder mehrere Obertypen angeben, deren Felder und Eigenschaften auf den Untertypen übergehen. Die Felder und Eigenschaften werden *vererbt*. Alle in einem Programm definierten Verbundtypen bilden eine *Vererbungshierarchie*.

Einzelne Verbunde, die in einem Programm vorkommen, stehen in der Regel in einer Beziehung zueinander, die durch Referenzen, die in den Verbunden gespeichert sind, ausgedrückt werden.

Definition 2.5.3 (Datenstruktur) *Eine Datenstruktur ist eine Ansammlung von Verbunden die durch Referenzen in Verbindung stehen.*

Datenstrukturen sind zum Beispiel Bäume, Listen oder Graphen, die mit Referenzen implementiert sind. In Programmen gehaltene Daten stehen häufig durch Referenzen miteinander in Verbindung, ohne dass der Programmierer gezielt eine Datenstruktur implementiert hat. In diesem Fall ist die Datenstruktur dem Programmierer nicht bewusst und nicht von anderen Aspekten des Programms gekapselt. Dadurch ist eine händische Optimierung durch den Programmierer noch aufwendiger. Solche Datenstrukturen können jedoch auch Pufferprobleme aufwerfen. Wir betrachten hier übergreifend alle Datenstrukturen, die anhand Referenzen zwischen Verbunden durch den Übersetzer erkennbar sind. Ein Programm, das vorrangig auf solche Datenstrukturen zugreift, nennen wir eine *Zeigeranwendung*.

Die Zugriffsmuster in Zeigeranwendungen sind weitaus vielfältiger als in numerischen Anwendungen. So sind laufzeitrelevante Iterationen eher mit Rekursionen oder Schleifen mit dynamischen Schleifengrenzen formuliert als mit Zählschleifen. Die Besuchsreihenfolge der Daten oder ein Muster der besuchten Speicheradressen lässt sich nicht direkt aus den Indizes einer Iteration ableiten. Daher lassen sich diese Algorithmen viel schlechter automatisch analytisch erfassen.

In der Regel werden Verbunde in der Hochsprache durch *Hochsprachtyp*en, kurz Typen, festgelegt. Dann können in einem Program nur Verbunde, die den Spezifikationen durch diese Typen genügen, vorkommen. Diese Typen können explizit oder implizit vereinbart werden. Der Typ eines Verbunds legt dann durch Auflistung der Typen der Felder und Sprachdefinitionen Eigenschaften der Verbunde des Typs fest. So wird in C durch das Konstrukt `struct` ein Verbund definiert, in dem per Sprachdefinition die Felder eine feste Reihenfolge haben. In Java wird durch `class` ebenfalls ein Verbund definiert. Hier ist die Reihenfolge der Felder nicht festgelegt. Die Sprachdefinition schreibt jedoch vor, dass man zur Laufzeit verschiedene Eigenschaften der Verbunde abfragen kann, zum Beispiel ihren Hochsprachtyp. Die Stapelschachtel ist ein implizit vereinbarter Verbund.

2.6 Pufferprobleme von Verbunden und Datenstrukturen

Dieses Kapitel betrachtet Eigenschaften dynamischer Verbunde und Datenstrukturen, die zu unnötigen Fehlern in Pufferspeichern führen können. Damit identifizieren wir die grundlegenden Probleme, die wir mit unserer Optimierung adressieren.

2.6.1 Probleme eines Verbunds

Ein einzelner Verbund stellt ein Leistungsproblem dar, wenn er eine hohe Wiederverwendung aufweist, die jedoch nicht als Lokalität im Pufferspeicher realisiert wird. Die Wiederverwendung eines Verbunds ergibt sich aus der Wiederverwendung der Zellen, die benötigt werden, um den Verbund zu speichern. Sie ergibt sich also aus der Wiederverwendung der Felder, die in den Zellen gespeichert sind.

Ein Feld hat eine hohe zeitliche Wiederverwendung, wenn im Laufe des Programms häufig auf das Feld zugegriffen wird. Wir nennen ein Feld mit hoher zeitlicher Wiederverwendung *pufferkritisch*, eines mit geringer zeitlicher Wiederverwendung *unkritisch*.

Enthält ein Verbund pufferkritische und unkritische Felder, begrenzt dies die räumliche Wiederverwendung. Liegt ein pufferkritisches Feld neben einem unkritischen Feld, wird bei einem Zugriff auf das pufferkritische Feld das unkritische mit der Speicherzeile ebenfalls in den Pufferspeicher geladen. Da auf das unkritische Feld selten zugegriffen wird, ist die räumliche Wiederverwendung zwischen diesen Feldern gering. Damit ist auch die Wahrscheinlichkeit, dass räumliche Lokalität zwischen diesen Feldern auftritt, gering. Liegen jedoch pufferkritische Felder nebeneinander, so dass sie gemeinsam in den Pufferspeicher geladen werden, ist die räumliche Wiederverwendung zwischen diesen Feldern im Vergleich hoch und damit auch die Wahrscheinlichkeit, dass diese als Lokalität ausgenutzt wird, deutlich höher.

Es ist also problematisch, wenn ein Verbund pufferkritische und unkritische Felder enthält, und diese gemischt angeordnet sind. Ein Verbund sollte nur Felder gleicher Pufferrelevanz enthalten. Wenn dennoch unkritische Felder in dem Verbund sind, sollten die Felder nach ihrer Pufferrelevanz sortiert sein, so dass, falls der Verbund eine Zeilengrenze überschreitet, die pufferkritischen Felder gemeinsam in eine Zeile fallen.

Die Zugriffsmuster eines Programms auf seine Verbunde können sich im Laufe des Programms ändern. Dadurch kann ein Feld in einem Teil des Programms pufferkritisch und in einem anderen Teil unkritisch sein. Dann treten an verschiedenen Stellen des Programms verschiedene Pufferprobleme auf.

Dies kann mit der Verwendung des Verbunds in einem bestimmten Algorithmus zusammenhängen. Werden zum Beispiel Adresseinträge in einer Liste nach dem Namen sortiert, sind die Listenfelder und das Feld mit dem Namen pufferkritisch, andere Felder jedoch unkritisch. Werden die Adressdaten jedoch ausgegeben, werden alle Felder verwendet und alle Felder haben die gleiche Pufferrelevanz.

Die Pufferprobleme können auch mit der Interpretation eines Verbunds in einer Vererbungshierarchie zusammenhängen. Wird ein Verbund im Kontext seines Obertyps verwendet, können die Felder, um die der Verbund den Obertyp ergänzt, nicht verwendet werden, da diese in diesem Kontext nicht bekannt sind. Diese Felder sind in diesem Kontext also unkritisch.

2.6.2 Probleme einer Datenstruktur

Die Probleme von Datenstrukturen manifestieren sich in der Beziehung einzelner Verbunde, aus denen eine Datenstruktur zusammengesetzt ist. Daher betrachten wir Wiederverwendung und Lokalität bezüglich ganzer Verbunde. Ein Verbund ist *pufferkritisch* bzw. *unkritisch*, wenn er vorrangig pufferkritische bzw. unkritische Felder enthält.

Räumliche Lokalität

Räumliche Wiederverwendung ist eine Voraussetzung um Lokalität zwischen zwei Verbunden zu erreichen. Räumliche Wiederverwendung zwischen Verbunden wird nur dann zu Lokalität, wenn diese mit einem geringen Zugriffsabstand vom Programm verwendet werden.

Auf einen dynamisch allozierten Verbund kann nur dann zugegriffen werden, wenn vorher eine Referenzvariable, das heißt, eine Variable die eine Referenz auf diesen Verbund enthält, gelesen wurde. Diese Referenzvariable kann eine lokale oder globale Variable des Programms sein. Sie kann jedoch auch ein Feld eines anderen Verbunds sein. Wird auf einen dynamisch allozierten Verbund zugegriffen, muss es also eine Variable geben, die die Referenz auf den Verbund enthält und auf die mit einem kurzen Zugriffsabstand zugegriffen wurde.

Sind alle Referenzvariablen auf einen Verbund unkritisch, kann der Verbund selbst nicht pufferkritisch sein.

Ist eine Referenzvariable jedoch pufferkritisch, und wird nach Zugriffen auf diese Variable mit der geladenen Referenz auf den referenzierten Verbund zugegriffen, ist auch dieser Verbund pufferkritisch. Da auf die Variable und den Verbund mit einem geringen Zugriffsabstand zugegriffen wird, kann räumliche Wiederverwendung leicht durch den Pufferspeicher als Lokalität ausgenutzt werden. Liegen die Variable und der Verbund in der gleichen Pufferzeile, besteht räumliche Wiederverwendung. Dies ist für lokale und globale Variablen unmöglich, da diese in anderen Speicherregionen angelegt werden als dynamisch allozierte Verbunde. Referenziert ein Verbund jedoch einen anderen Verbund, kann hier räumliche Wiederverwendung auftreten. Leider beachten gewöhnliche Allokationsalgorithmen dies nicht, so dass Verbunde zufällig im Speicher platziert werden. Dies kann durch eine Optimierung verbessert werden.

Pufferkritische Referenzfelder in einem Verbund sind also Indikatoren dafür, dass durch Herstellung räumlicher Wiederverwendung die Lokalität einer Datenstruktur erhöht werden kann.

Wird ein pufferkritisches Referenzfeld häufig geschrieben, besteht keine dauerhafte Beziehung zwischen zwei Verbunden, für die sich gezielt Wiederverwendung herstellen lässt. Trotzdem weist das pufferkritische Referenzfeld darauf hin, dass Verbunde der beiden involvierten Typen häufig gemeinsam verwendet werden. Das Verhältnis zwischen schreibenden und lesenden Zugriffen auf eine Referenzvariable charakterisiert also das Pufferverhalten eines Verbunds in einer Datenstruktur.

Zeitliche Lokalität

Kann eine Datenstruktur während ihrer Lebenszeit im Puffer gehalten werden, haben alle Verbunde in der Datenstruktur zeitliche Lokalität. Optimierungen der zeitlichen Lokalität von Datenstrukturen sind also nur sinnvoll, wenn die Datenstruktur oder Teile davon aus dem Puffer verdrängt werden.

Dies kann durch Konflikte mit Verbunden aus anderen Datenstrukturen passieren. Da dynamische Allokation Adressen eher zufällig vergibt, sind systematische Probleme durch Konflikte in der Adressabbildung unwahrscheinlich. Kapazitätsfehler treten jedoch systematisch auf, sobald die Daten eines Programmes die Puffergröße überschreiten. Große Datenstrukturen können

also ihre zeitliche Wiederverwendung schlecht als Lokalität im Puffer realisieren.

Große Datenstrukturen können nicht in sequentiellem Programmcode aufgebaut werden, da die Größe der Datenstruktur dann linear von der Programmgröße abhängt. Sie werden in Schleifen oder Rekursionen alloziert. Hohe zeitliche Wiederverwendung wird durch Zugriffe auf eine Datenstruktur in Schleifen oder Rekursionen erreicht.

Zeitliche Wiederverwendung setzt eine lange Lebenszeit einzelner Verbunde voraus. Die Lebenszeit eines einzelnen Verbunds ist statisch sehr schwer abzuschätzen. Dies gilt insbesondere für Programmiersprachen, in denen es keine explizite Deallokation gibt.

Datenstrukturen, deren zeitliche Lokalität durch Optimierung verbessert werden kann, sind also groß und langlebig. Ein Indikator hierfür ist die Allokation und Verwendung in Schleifen und Rekursionen.

2.6.3 Einzelne Verbunde

Einzelne dynamisch lebendige Verbunde können zur Übersetzungszeit schwer identifiziert werden. In der Regel ist dies für dynamisch allozierte globale Variablen möglich. Werden Verbunde dynamisch in Schleifen oder Rekursionen alloziert, können wir nicht alle einzelnen Verbunde unterscheiden. Haldenanalysen wie in [Tra01] unterscheiden Gruppen von Verbunden eines Typs durch einen Namen. Ungenauer ist es, Verbunde nur nach ihrem Typ zu unterscheiden.

Können wir einzelne Verbunde nicht unterscheiden, können wir auch Pufferprobleme einzelner Verbunde nicht feststellen. Wir abstrahieren immer für eine Menge Verbunde, die unter gleichen Gesichtspunkten verwendet werden.

Verschiedene Verbunde gleichen Typs müssen gegenüber dem Programm gleichartig erscheinen. Ein Verbund muss an jeder Stelle im Programm, an der Verbunde dieses Typs vorkommen können, korrekt behandelt werden können. Dies gilt auch für Verbunde in Vererbungshierarchien. Daher müssen Optimierungen, die innerhalb eines Verbunds angreifen, auf alle Verbunde eines Typs (bzw. in einer Vererbungshierarchie) angewandt werden.

Optimierungen, die die relative Position zwischen Verbunden betreffen, können jedoch feingranularer angewandt werden, zum Beispiel für ausgewählte Namen. Dazu werden dann Informationen über die Pufferprobleme bezüglich dieser Namen benötigt.

2.7 Voraussetzungen dieser Arbeit

2.7.1 Anforderungen an die Programme

Optimal für die Pufferoptimierung von Zeigeranwendungen sind Programme, die in vollständig typisierten Sprachen implementiert sind. Sprachen mit Zeigerarithmetik erschweren das Erkennen aller Zugriffe auf einen Verbund. Programme, auf die sinnvoll Pufferoptimierungen angewandt werden, müssen große Datenstrukturen enthalten, die in Schleifen oder Rekursionen bearbeitet werden.

Diese Datenstrukturen sollten nur in der übersetzten Einheit sichtbar sein. Gelangen die Datenstrukturen in externe Programmteile, die nicht übersetzt werden, kann der Übersetzer durch Analysen kein vollständiges Bild der Verwendung der Datenstruktur erlangen. Da die Datenstrukturen zu den Verwendungen in den externen Programmteilen kompatibel sein müssen, sind die Optimierungsmöglichkeiten eingeschränkt.

Wir nehmen an, dass die übersetzten Programme an einer für den Übersetzer erkennbaren Stelle initial aufgerufen werden. Der Übersetzer kennt die Laufzeitumgebung und kann die Effekte von Aufrufen in die Laufzeitumgebung abschätzen. Sind jedoch weitere Teile eines Programms unbekannt, kann der Übersetzer einige Optimierungen nicht anwenden, und die Analysen können zu ungenauen Erkenntnissen kommen.

2.7.2 Anforderungen an die Programmdarstellung

Um die oben aufgeführten Probleme von Verbunden und Datenstrukturen zu erkennen, in der Optimierung zu erfassen und die Verbunde dementsprechend zu transformieren, können wir einige Anforderungen an unsere Programmdarstellung ableiten.

Wir unterscheiden die Pufferprobleme einzelner Felder. Die Pufferprobleme ergeben sich aus den Zugriffen auf die Felder. Die Programmdarstellung muss also die Felder der Verbunde unterscheiden können, sowie bei Zugriffen angeben, auf welche Felder zugegriffen wird. Insbesondere sollten Felder, die Referenzen enthalten von anderen Feldern unterschieden werden können. Um zu wissen, was für ein Verbund referenziert wird, muss der Zieltyp einer Referenz bekannt sein.

Da Vererbungshierarchien die Transformationsmöglichkeiten der Verbunde einschränken, sollten wir diese ebenfalls in der Darstellung repräsentieren.

Um Verwendungsunterschiede von Feldern im Vererbungskontext feststellen zu können, muss dieser in der Darstellung erkennbar sein. Insbesondere muss darstellbar sein, ob ein Zugriff auf ein vererbtes Feld nur im Kontext eines bestimmten Untertyps stattfindet.

Man muss der Programmdarstellung ansehen, wo Typkompatibilität gefordert ist. Man muss leicht ableiten können, wann die von der Hochsprache geforderte Kompatibilität für konkrete Programme eingeschränkt werden kann.

Kapitel 3

Klassifikation bekannter Pufferoptimierungen

Pufferoptimierungen können nach drei orthogonalen Kriterien klassifiziert werden.

Erstens können wir Optimierungen für den Befehls- oder den Datenpuffer unterscheiden. Da auf Befehle und Daten ganz unterschiedlich zugegriffen wird, gibt es ganz unterschiedliche Optimierungsstrategien. Diese Arbeit konzentriert sich auf Datenpufferoptimierungen.

Zweitens gibt es eine Vielzahl verschiedener Optimierungsstrategien. Zunächst muss man Optimierungen, die Fehler vermeiden von solchen unterscheiden, die den Schaden, der durch das Auftreten von Fehlern entsteht, verringern.

Fehler kann man vermeiden, indem man den Algorithmus des Programms ändert, oder indem man die Daten gezielt im Speicher anordnet. In beiden Fällen kann man Verfahren anwenden, die Kapazitäts- oder Konfliktfehler optimieren. Durch Datenanordnungsoptimierungen kann man auch obligatorische Fehlzugriffe vermeiden, da diese Optimierungen die Wiederverwendung beeinflussen können.

Der Schaden, der durch Fehler entsteht, kann vermieden werden, indem man mit einem speziellen Befehl, einem Vorladebefehl, die Zeile rechtzeitig lädt. Eine andere Möglichkeit ist, die Befehle so anzuordnen, dass der Prozessor weitere sinnvolle Arbeit tun kann, bis der Zugriff erledigt ist.

Schlussendlich kann man Optimierungen, die statische Referenzstellen optimieren, von solchen unterscheiden, die dynamische Referenzstellen optimieren.

3.1 Fehler Vermeiden

Um Fehler zu vermeiden, muss man die Lokalität erhöhen. Dazu kann man die vorhandenen Wiederverwendung des Programms besser ausnutzen. Man kann jedoch auch die Wiederverwendung erhöhen und dadurch mehr Möglichkeiten für Lokalität schaffen.

3.1.1 Optimierung durch Restrukturierung der Berechnungen

Durch die Restrukturierung eines Algorithmus wird die Reihenfolge, in der Zugriffe ausgeführt werden, geändert. Die Anzahl der Zugriffe auf eine gegebene Zelle bleibt unverändert, und damit auch die Wiederverwendung des Programms. Die Zugriffe werden jedoch in anderer Reihenfolge ausgeführt und von anderen Referenzstellen initiiert. Ist die Optimierung erfolgreich, wird der Zugriffsabstand auf gleiche Zellen oder auch Zeilen geringer, wodurch sich die Lokalität erhöht. Alternativ kann der Zugriffsabstand auf Zellen mit Konflikten gezielt erhöht werden.

Statisch vorhersagbare Zugriffe

Um einen Algorithmus automatisch zu verändern, muss man diesen sehr genau analysieren können. Dies ist für numerische Anwendungen, die statisch vorhersagbare Zugriffe enthalten, gegeben (Kap. 2.5). Daher sind hier vielfältige Optimierungen bekannt. Diese werden in Tabelle 3.1 zusammengefasst.

Schleifentransformationen (*Permutation*, *Umkehr*, *Neigen*) ändern die Laufvariablen der Schleifen so, dass die einzelnen Schleifenrümpfe für jede Indexkombination in einer anderen Reihenfolge ausgeführt werden. Dies verbessert die Lokalität, indem vorrangig Kapazitätsfehler vermieden werden. Für Schleifentransformationen existieren geschlossene mathematische Beschreibungen.

Durch *Verschmelzen* und *Teilen* aufeinander folgender Schleifen wird die Lokalität durch Vermeiden von Kapazitäts- und Konfliktfehlern erhöht. Verschmelzen kann die Wiederverwendung innerhalb einer Schleife verbessern.

Das *Kacheln* von Schleifen bewirkt, dass mehr Berechnungen auf einem Teil einer großen Reihung ausgeführt werden, bevor zum nächsten Teil der Reihung übergegangen wird. Dies reduziert ebenfalls Kapazitätsfehler. Durch

Optimierung	Einzelne Arbeiten	Rahmenwerke				
		[WL91]	[ST92]	[KP93]	[Li94]	[CMT94]
Verschmelzen	[KM93, GOST93, SM98]					x
Teilen				x		x
Permutation	[AK84]	x	x	x	x	x
Kacheln (Kap.)	[Wol87, GJG88, CL95a]	x	x	x		x
Kacheln (Kon.)	[LRW91, Ess93, CM95]					
Umkehr		x	x	x	x	x
Neigen		x	x	x	x	x

Tabelle 3.1: Pufferoptimierungen für Schleifenalgorithmen

schlechtes Kacheln können jedoch extreme Konfliktfehler auftreten. Daher beschäftigen sich hier Arbeiten speziell mit dem Vermeiden von Konfliktfehlern.

In vielen Arbeiten wird versucht, unter Anwendung mehrerer dieser Optimierungen möglichst pufferoptimale Schleifen zu erzeugen. Dazu werden Rahmenwerke formuliert. Jüngere Arbeiten [SM98] beschäftigen sich vorrangig mit Randproblemen dieser bekannten Optimierungen, oder mit der Kombination mit anderen Übersetzeroptimierungen. Schleifenoptimierungen sind häufig für spezielle Algorithmen, wie zum Beispiel Matrixmultiplikation, besonders effizient.

Dynamische Zugriffe

Algorithmen in Zeigeranwendungen sind durch dynamische Zugriffe gekennzeichnet. Da diese schwer zu analysieren sind (Kap. 2.5), existieren hier sehr wenige Arbeiten, die sich meistens auf die Analyse und Verbesserung der Implementierung standardisierter Algorithmen beschränken [LL96, LL97, LFL99]. Diese helfen damit einem Programmierer, einen Algorithmus unter Gesichtspunkten der Pufferleistung von Hand zu verbessern.

3.1.2 Datenanordnung Optimieren

Durch die Optimierung der Datenanordnung im virtuellen Speicher kann man die Wiederverwendung und Lokalität erhöhen. Die Abbildung der virtuellen Adresse einer Zelle auf eine physikalische Adresse in einem Puffer kann vom Übersetzer nicht beeinflusst werden. Daher wird die Abbildung der Daten

auf die Puffer beeinflusst, indem man günstige virtuelle Adressen wählt.

Variablen werden auf feste Zellen abgebildet. Diese Abbildung ist in der Regel für die Lebenszeit der Variablen gültig. Bildet der Übersetzer zwei gemeinsam verwendete Variablen auf Zellen ab, die in verschiedenen Zeilen liegen, haben diese Zellen keine gemeinsame räumliche Wiederverwendung. Durch Abbildung der Zellen auf eine gemeinsame Zeile kann räumliche Wiederverwendung erreicht werden, die dann zu Lokalität führen kann.

Der erste Zugriff auf jede dieser Variablen ist ein potentieller obligatorischer Fehler. Haben die Variablen disjunkte Lebenszeiten, können sie auf die gleiche Zelle abgebildet werden. Dadurch kann der obligatorische Fehlzugriff auf die später lebende Variable vermieden werden. Es wurde zeitliche Wiederverwendung geschaffen, die zu Lokalität führen kann.

Statisch vorhersagbare Zugriffe

Da sich die Datenanordnung in numerischen Anwendungen genauso gut durch Analysen erfassen lässt wie Algorithmen, existieren auch hier vorrangig Arbeiten für numerische Anwendungen. Tabelle 3.2 fasst Arbeiten in diesem Bereich zusammen.

Die Optimierung *Kopieren* wählt einen Teil der Datenmenge eines Programms aus. Für eine Berechnung mit Pufferproblemen kopiert es diese Datenmenge in eine eigene Speicherregion im virtuellen Speicher in einer Reihenfolge, die für die Zugriffsfolge der Berechnung günstig ist. Nach der Berechnung werden die Daten an die ursprünglichen Speicherplätze zurück kopiert oder verworfen. Da der Gewinn an Pufferleistung die Kosten des Kopierens rechtfertigen muss, ist Kopieren nur bei extremen Pufferproblemen sinnvoll. Kopieren wird typischerweise mit Kacheln eingesetzt, um Konflikte, die durch ungünstige Kachelgrößen entstehen, zu vermeiden.

Abgesehen von den neu eingeführten Zugriffen für die Kopieroperationen erhöht Kopieren die räumliche Wiederverwendung und sorgt dafür, dass bestehende Wiederverwendung durch Vermeidung von Konflikten als Lokalität ausgenutzt wird.

Füllen fügt unbenutzte Felder zwischen oder innerhalb von Reihungen ein. Durch die Füllungen kann die Anfangsadresse von Reihungen oder der Adressabstand von Feldern beeinflusst werden. Dadurch können Konflikte zwischen zwei Feldern, die gleichzeitig benötigt werden, vermieden werden. Füllen erhöht die Wiederverwendung nicht. Es kann sie am Übergang zu den Füllungen geringfügig verschlechtern.

Optimierung	Arbeit
Kopieren	[LRW91, BJWE91, TGJ93]
Füllen	[BCJ+94, PNDN97, RT98]
Transposition	[CL95b]
Mischen	[LW94]
Packen	[LW94]

Tabelle 3.2: Arbeiten für statisch vorhersagbare Zugriffe

Reihungstransformationen (*Transposition*, *Reversion*) funktionieren ähnlich wie Schleifentransformationen. Sie vertauschen die Anordnung oder Richtung der Dimensionen mehrdimensionaler Reihungen. Reihungstransformationen lassen sich wie Schleifentransformationen mathematisch beschreiben. Da Reihungstransformationen weniger durch Datenabhängigkeiten eingeschränkt werden als Schleifentransformationen, aber die gleichen Optimierungsmöglichkeiten bieten, sind sie eine gute Alternative wenn Datenabhängigkeiten die benötigten Schleifentransformationen verhindern.

Mischen verzahnt mehrere Reihungen zu einer großen Reihung, so dass die Felder der ursprünglichen Reihungen in fixen Abständen in der verzahnten Reihung vorkommen. Dadurch können gemeinsam verwendete Felder in einer Zeile platziert werden. Dadurch wird die räumliche Wiederverwendung so geändert, dass diese besser als Lokalität ausgenutzt wird.

Packen reduziert die Größe der Felder. Dadurch fallen mehr Felder in eine Zeile, was die räumliche Wiederverwendung erhöht.

Es ist teuer, die Anordnung einer Reihung während der Laufzeit zu ändern. Da verschiedene Schleifen verschiedene Datenanordnungen erfordern können, ist es schwer eine allgemein gute Anordnung zu finden. Mit diesem Problem haben wir uns in [GL01] beschäftigt.

Dynamische Zugriffe

Aufgrund der Freiheitsgrade in der Datenanordnung (Kap. 2.5) bei Zeigeranwendungen ist dies der häufigste Optimierungsansatz für dynamische Zugriffe.

[CHL99] schlagen zwei Techniken vor, um Verbunde relativ zueinander anzuordnen: *Anhäufen* (clustering) und *Färben* (coloring). Anhäufen platziert Verbunde, die wahrscheinlich gleichzeitig benötigt werden, in einer Pufferzeile. Dies kann zum Beispiel ein Knoten eines Baumes mit seinen Nachfahren

sein. Dadurch wird räumliche Wiederverwendung hergestellt. Färben organisiert Pufferzeilen so, dass häufig verwendete Zeilen konfliktfrei auf einen Teil des Pufferspeichers abgebildet werden können. Alle anderen Pufferzeilen werden so angeordnet, dass sie auf den Rest des Pufferspeichers abgebildet werden. Der Artikel präsentiert zwei Werkzeuge, die dies unterstützen. Das eine steuert die Allokation neuer Verbunde, das andere kopiert die Datenstruktur um, um ein Layout mit der oben erklärten Strukturierung herzustellen. Beide Werkzeuge müssen vom Programmierer in das Programm eingeflochten werden.

Nur wenn mehrere Verbunde in eine Pufferzeile passen und die Datenstruktur sich nicht all zu schnell verändert sind diese Techniken effizient. Ansonsten wird die räumliche Wiederverwendung zu schnell zerstört, um die zusätzlichen Kosten zu amortisieren.

In [CDL99] wird diese Arbeit auf das interne Layout der Verbunde ausgedehnt. [CDL99] führen *Verbundteilen* (structure splitting) und *Feldumordnen* (field reordering) ein, um die Anzahl der Verbunde, die in eine Pufferzeile passen, zu erhöhen. Dazu werden die Felder eines Verbunds nach ihrer Zugriffshäufigkeit und -reihenfolge sortiert und selten benutzte Felder abgetrennt. Der Zugriff auf die abgetrennten Felder erfolgt mittels einer Indirektion über eine im ursprünglichen Teil gespeicherte Referenz. Dies bedeutet einen Laufzeitzuschlag und zusätzlichen Speicherverbrauch. Auch hier wird ein Werkzeug, das den Programmierer unterstützt, vorgestellt.

[KF00] schlagen Verbundteilen und Feldumordnen vor, um die Lokalität typsicherer Programme zu verbessern. Um Verbundteilen zu implementieren verwenden sie im Gegensatz zu [CDL99] keine Referenzen um auf den abgetrennten Teil zu verweisen. Sie modifizieren die Speicherverwaltung so, dass sie für einen Verbund mehrere gleich große Blöcke in einem festen Abstand alloziert. Da sie sich auf typsichere Programme beschränken, können sie die Optimierung im Übersetzer automatisieren. Zugriffshäufigkeiten ermitteln sie durch Messungen.

[SZ97] setzen Färben automatisch ein. Durch Messungen finden sie die häufig benutzten Verbunde. Ihre Daten basieren auf Binärcodes, und sie ändern lediglich die Funktion der Speicherallokation ohne jeglichen Eingriff in einen Übersetzer. Daher ist es problematisch, dem Allokator zu kommunizieren, welche Verbunde speziell behandelt werden sollen. Hierzu werden einige Heuristiken vorgeschlagen. Diese werden anhand der Messdaten automatisch abgestimmt. Durch den Verzicht auf einen Eingriff in einen Übersetzer werden hier viele Möglichkeiten vergeben.

[GTZ98] präsentieren *Typanhäufen* als eine echte Übersetzeroptimierung. Typanhäufen platziert Verbunde eines bestimmten Typs auf eine eigene Halde. Dies erhöht die Lokalität dieser Verbunde, wenn gemeinsam auf diese Verbunde zugegriffen wird. Die Optimierung basiert auf einer Analyse des Typgraphen des Programms. Die Analyse findet Behältertypen, die die *Sandwich* Eigenschaft erfüllen: Die Verbunde, die auf einer eigenen Halde platziert werden, sind das Innere des Sandwiches. Die untere Schicht sind die Verbunde, die in den Behältern gehalten werden. Die obere Schicht sind Verbunde, durch die auf die Behälterobjekte zugegriffen werden kann. Die Optimierung basiert auf der Annahme, dass Traversierungen und damit häufige Speicherzugriffe vornehmlich auf diesen Behältertypen stattfinden. Dies ist eine sehr grobe Heuristik, die die Optimierung für alle Sandwichtypen anwendet, unabhängig von deren Bedeutung für das Programm. Wir greifen den Optimierungsansatz von [GTZ98] auf, ersetzen aber die Heuristik und kombinieren Typanhäufen mit weiteren Optimierungen.

[LA05, Lat05] definieren eine Programmdarstellung namens LLVM die ähnlich unserer Darstellung ([Lin02]) eine zielsprachnahe Programmdarstellung eng mit Typinformationen kombiniert. Mit einer effizienten interprozeduralen Analyse identifizieren sie isolierte Datenstrukturen in nicht typsicheren Programmen. Die Effizienz wird durch großzügige Unifikation und Flussinsensitivität erreicht. Die erkannten isolierten Datenstrukturen werden mit *Pool Allocation* optimiert. Dabei werden sie mit einem speziellen Allokator ähnlich unserem Typanhäufen in eine eigene Speicherregion alloziert. Für Verbunde die in dieser Speicherregion alloziert werden, werden weitere Optimierungen, wie zum Beispiel Anhäufen, vorgeschlagen. Diese Arbeit kann kleinere Datenstrukturen identifizieren als unsere Analyse, diese unterliegen aber mehr Restriktionen, so werden zum Beispiel Alias- und Lebenszeitinformationen benötigt. Die Optimierung versucht nicht, gezielt pufferkritische Datenstrukturen zu optimieren, sondern optimiert alle als optimierbar erkannten Datenstrukturen.

[WLM92] untersuchen die Effekte von Speicherbereinigung auf die Pufferleistung. Standardalgorithmen wie *Copying* und *Mark-sweep* fassen den gesamten benutzten Speicher innerhalb kürzester Zeit an. Dies belastet die Speicherhierarchie enorm und erzeugt viele Kapazitätsfehler, da die komplette Datenmenge eines Programms in der Regel nicht in den Puffer passt.

[WLM92] schlagen vor, Copying Techniken zu verwenden, die einzelne Generationen von dynamischen Verbunden bereinigen. Dies verringert die Größe der Datenmenge einer Bereinigung und für einige Verbunde den Zeitabstand

zwischen den Phasen. Mit Simulationen untermauern sie diesen Vorschlag.

[SGF+02] optimieren ebenfalls die Lokalität von Bereinigungsalgorithmen. In [SGBS02] werden *prolific types* eingeführt. Dies sind Typen, von denen zur Laufzeit überdurchschnittlich viele Instanzen alloziert werden. Diese Typen werden durch Laufzeitmessungen identifiziert. Sie diskutieren auch, Informationen über die Allokationshäufigkeit vom Programmierer an das Programm zu annotieren. [SGF+02] wendet für Verbunde solcher ‚prolific types', die sich gegenseitig referenzieren, Anhäufen (co-allocation) an. Sie verwenden keine Daten über die Häufigkeit der Zugriffe auf diese Verbunde. Über die anzuhäufenden Verbunde wird durch den Allokator entschieden. Dies bedeutet zusätzlichen Laufzeitaufwand.

[Sin02] benutzt gemessene Allokations- und Feldzugriffshäufigkeiten um die Traversierungsreihenfolge der Speicherbereinigung zu beeinflussen. Er untersucht verschiedene Kopierreihenfolgen für kopierende Speicherbereiniger bezüglich ihere Pufferleistung.

3.2 Fehlerkosten verringern

Fehler lassen sich nicht vollständig vermeiden. Obligatorische Fehler sind prinzipiell sehr schwer vermeidbar. Anwendungen die auf großen Datenmengen operieren und beliebige Zellen miteinander kombinieren, bedingen immer Kapazitätsfehler. Hier setzen Optimierungen, die die Kosten von Fehlern verringern, an.

Ein Großteil der Techniken zum Verringern der Nachteile durch Fehler ist unabhängig vom Referenztyp, da sie auf einzelne Referenzstellen angewandt werden. Oft benötigt man jedoch Information, ob eine Ladeoperation häufig ein Fehlzugriff ist. Dies ist für vorhersagbare Referenzstellen leichter zu ermitteln. Oft kann diese Information direkt aus einer Optimierung zur Verringerung der Fehlzugriffe generiert werden.

Trotzdem sind in diesem Bereich, vor allem für Vorladen, verhältnismäßig viele Arbeiten für dynamische Referenzstellen zu finden. Dies begründet sich darin, dass das Einfügen von Vorladebefehlen keine Korrektheitsprobleme aufwirft und wohl auch im Fehlen von Techniken zum Vermeiden von Fehlzugriffen für diese Referenzstellen. Im Folgenden diskutieren wir Techniken für dynamische Referenzstellen.

3.2.1 Vorladen

Vorladen verwendet spezielle Befehle, um eine Zeile vor dem eigentlichen Zugriff in den obersten Puffer zu laden. Diese Befehle sollten vorsichtig platziert werden, da sie die Speicherhierarchie zusätzlich belasten. Unnötiges Vorladen kann zum Beispiel einen Engpass auf dem Bus zum Speicher verursachen, wodurch notwendige Ladeoperationen verzögert werden. Daher sollte Vorladen einer bereits geladenen Zeile vermieden werden. Weiter muss man darauf achten, dass eine vorgeladene Zeile nicht vor der eigentlichen Nutzung wieder aus dem Puffer verdrängt wird.

[LSKR95] behandeln Vorladen bei Methodenaufrufen in Zeigeranwendungen. Sie verwenden eine einfache Heuristik, um Vorladebefehle einzufügen. Sie nehmen an, dass Methoden auf die Verbunde, deren Adresse sie als Parameter mitgegeben bekommen, zugreifen werden. Daher fügen sie Vorladebefehle für alle Adressen in den Argumenten vor dem Methodenaufruf ein. Dies ist offensichtlich sinnvoll für Zugriffsroutinen auf Felder von Verbunden. Mit Simulationen zeigen sie, dass die Anzahl der Fehlzugriffe beim Laden verringert werden kann.

[LM96, LM99a] betrachten Vorladen für rekursive Datenstrukturen wie Bäume, Listen und Graphen. Sie erkennen rekursive Datenstrukturen an der Typdeklaration, und erkennen Iterationen über die Datenstrukturen daran, dass Referenzstellen in Schleifen oder Selbstrekursionen induktiv dereferenziert werden. Sie laden die induktiv dereferenzierte Referenz vor. Alternativ speichern sie Referenzen auf Knoten, die einige Schritte später zugegriffen werden, im aktuellen Knoten, um diese Referenz zum Vorladen zu verwenden.

[SAG+01] schlagen eine andere Technik vor, um rekursive Datenstrukturen, insbesondere Listen, vorzuladen. Sie erkennen Iterationen ähnlich wie [LM96, LM99a], laden aber eine Adresse mit einem kleinen Versatz zu der Adresse des aktuellen Knotens vor. Dabei verlassen sie sich auf die Annahme, dass die Knoten sequentiell alloziert wurden und auch so im Speicher liegen.

[Cah02, CM01] präsentieren eine Analyse zur Übersetzungszeit für Vorladen in Zeigeranwendungen. Die Analyse findet Schleifen und Rekursionen, in denen über eine Datenstruktur iteriert wird. Dazu setzen sie intra- und interprozedurale Datenflussanalysen ein. Dadurch finden sie weitaus mehr Iterationen als die vorher diskutierten Ansätze. Zum Vorladen setzen sie die gleichen Techniken wie [LM96, LM99a] ein.

3.2.2 Ladelatenzzeitsensitive Codeanordnung

Hier wird versucht, Ladebefehle im Programm so zu platzieren, dass zwischen dem Laden und der Verwendung der Daten genügend Zeit bleibt, um die Latenz eines Fehlers zu überbrücken. Gelingt dies, kann ein Vorladebefehl eingespart werden. Allerdings erhöht diese Technik den Registerdruck.

Codeanordnung mit Listen Codeanordnung mit Listen (List Scheduling) ist die grundlegende Technik der Codeanordnung. Aus einer Liste ausführbarer Befehle wird nach einer Heuristik einer für die Ausführung ausgewählt und in die Anordnung eingefügt. Die Heuristik berücksichtigt in der Regel die Latenzzeit der Operationen, kann jedoch jede beliebige Information verwenden, zum Beispiel Fehlzugriffshäufigkeiten.

[KE93] schlagen *Balancierte Codeanordnung* (Balanced Scheduling) vor. Dieser Algorithmus balanciert die verfügbare Parallelität auf in Frage kommende Anordnungskandidaten. Der dadurch zu Stande kommende Wert wird dann in der Heuristik für die Listenauswahl verwandt. Die Stärke des Algorithmus ist es, bei unbekannten Latenzzeiten eine gleichmäßige Anordnung zu erreichen.

[LE95] kombinieren den Algorithmus von [KE93] mit einer Lokalitätsanalyse für vorhersagbare Referenzstellen. Sie weisen, abhängig von der Fehlzugriffwahrscheinlichkeit, Ladebefehlen die Latenz des ersten oder zweiten Puffers zu. Dadurch wird Parallelität vorrangig an Ladebefehle mit hoher Fehlzugriffwahrscheinlichkeit verteilt. Sie implementieren diesen Algorithmus in einem hoch optimierenden Übersetzer. Simulationen des produzierten Codes zeigen das Potential des Anordnungsalgorithmus.

In [LMT00] verwenden wir den Algorithmus und Übersetzer von [LE95]. Wir ersetzen jedoch die Lokalitätsanalyse durch Daten, die durch Laufzeitmessungen gewonnen werden. Dadurch können weitaus mehr Ladebefehle mit Daten versehen und insbesondere auch dynamische Referenzstellen behandelt werden.

Software Pipelining Eine weitere Anordnungstechnik, für die es ladelatenzzeitsensitive Algorithmen gibt, ist *Software pipelining*. Hier werden mehrere Iterationen einer Schleife parallel angeordnet. Daher bewirkt ein langer kritischer Pfad durch lange Latenzen keine schlechte Anordnung sondern nur eine hohe Zahl paralleler Iterationen. Dies erhöht jedoch den Registerdruck und die Anzahl von Registerkopierbefehlen. Für Ladebefehle kann

dadurch aber eine beliebige Latenzzeit eingesetzt werden. Verwendet man jedoch nur Latenzzeiten für Fehlzugriffe, müssen viele Iterationen parallel angeordnet werden. Verwendet man nur Latenzzeiten für Puffertreffer, werden die Latenzen der Fehlzugriffe ungenügend versteckt. [DCS96, DCS97] und [SG97] verwenden selektiv Latenzen für Fehlzugriffe. Die Information über Fehlzugriffe schöpfen sie aus Analysen für vorhersagbare Referenzstellen.

3.3 Pufferoptimierungen in der Übersetzerarchitektur

Pufferoptimierungen können an verschiedenen Stellen in der Übersetzerarchitektur angebracht werden.

Optimierungen, die Fehler zu vermeiden suchen, werden in einer frühen Phase der Übersetzung ausgeführt. Datenstrukturen und Algorithmen sind dann am einfachsten zu verändern, wenn sie noch den Einschränkungen der Quellsprache entsprechen, so dass diese ausgenutzt werden können.

Die ersten Optimierungen für numerische Anwendungen schlugen Veränderungen von Schleifenschachteln vor und richteten sich an den Programmierer. Diese Techniken konnten einfach durch Quelltransformationen automatisiert werden. Dies trifft auch für Anordnungsoptimierungen numerischer Anwendungen zu. Werden Schleifenoptimierungen auf der Zwischensprache eines Übersetzers ausgeführt, ist es vorteilhaft, Kenntnisse über die Schleifenstruktur, die sich anhand des abstrakten Syntaxbaums leicht erkennen lassen, für die Optimierung zu konservieren. Dies ist insbesondere sinnvoll, wenn die Hochsprache leicht analysierbare Schleifenkonstrukte enthält, welche zum Beispiel die Schleifengrenzen syntaktisch erkennbar benennen.

Optimierungen für Zeigeranwendungen mit expliziter Zeigerarithmetik werden in der Regel als Quelltransformationen durch den Programmierer ausgeführt. Verzichtet eine Hochsprache jedoch auf explizite Zeigerarithmetik, wie zum Beispiel Java, sind diese Quelltransformationen automatisiert, oder werden während der Übersetzung auf der Zwischensprache durchgeführt.

Anordnungsoptimierungen müssen oft in die Speicherverwaltung integriert werden, da diese zur Laufzeit des Programms die relative Anordnung einzelner Programmdaten beeinflusst. Dies ist insbesondere bei Zeigeranwendungen sinnvoll, um die relative Platzierung verschiedener Verbunde zu kontrollieren. Für Hochsprachen mit expliziter Speicherverwaltung wird diese

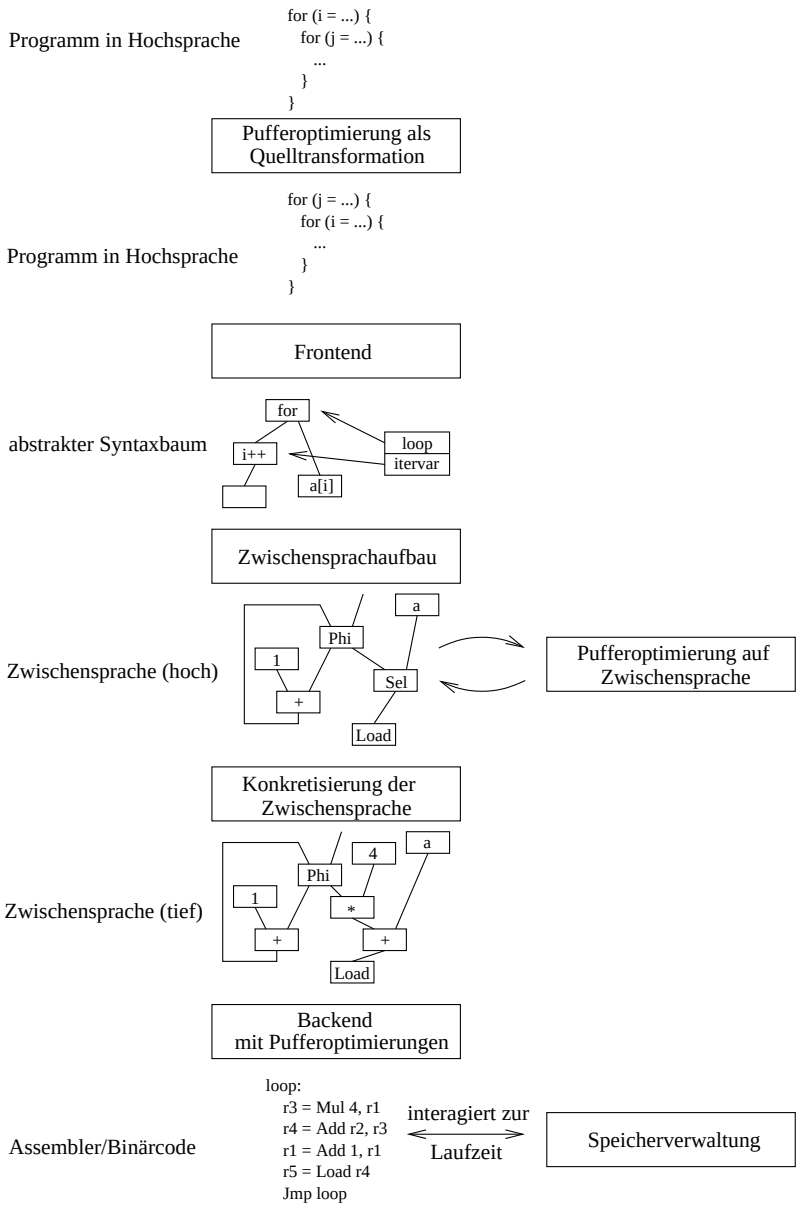

Abbildung 3.1: Pufferoptimierungen in der Übersetzerarchitektur

um neue Verwaltungsroutinen ergänzt. Sieht eine Hochsprache automatische Speicherverwaltung vor, können Anordnungsoptimierungen in den Speicherbereiniger integriert werden.

Optimierungen, die Fehlerkosten verringern, werden in einer späten Phase des Übersetzers ausgeführt. Hier werden einzelne Ladebefehle betrachtet. Diese Optimierungen hängen von spezifischen Eigenschaften der Zielmaschine ab, wie Puffergröße, Ladelatenz oder den vorhandenen Vorladebefehlen. Viele dieser Optimierungen werden während der Codeanordnung und Codeauswahl durchgeführt. Analysedaten, die zur Optimierung benötigt werden, können aber oft besser in frühen Phasen der Übersetzung berechnet werden.

3.4 Kritik

Diese Übersicht zeigt, dass die Forschung Pufferoptimierungen für dynamische Datenstrukturen stiefmütterlich behandelt. Optimierungen zur Reduktion von Fehlern in Zeigeranwendungen sind vorrangig Programmieranleitungen oder Werkzeuge zur Unterstützung des Programmierers. Pufferprobleme werden durch Messungen identifiziert. Im internationalen Umfeld existiert keine automatische Übersetzeroptimierung, die die Datenanordnung von Zeigeranwendungen optimiert. Es hat sich jedoch gezeigt, dass Ansätze, die Eingriffe des Anwendungsprogrammierers verlangen, nur in Spezialfällen angewendet werden. Dies liegt einerseits am zusätzlichen Lernaufwand zur Bedienung der Optimierung und andererseits an der zusätzlichen Entwicklungszeit. Die Ansätze, die hier beschrieben wurden, zeigen jedoch das Potential dieser Optimierungen. D.h, es existieren sinnvolle Transformationen; es fehlen aber die Analysetechniken, um diese Transformationen automatisch anzuwenden.

Kapitel 4

Eine geeignete Darstellung

Die in dieser Arbeit vorgestellten Optimierungen profitieren grundlegend von einer geeigneten Darstellung des übersetzten Programms im Übersetzer. Dies betrifft sowohl die Darstellung des Programmcodes als auch die Darstellung der Typinformation des Quellprogramms.

Im folgenden stellen wir die Zwischendarstellung Firm vor. Firm wurde ursprünglich in [Tra01] definiert. Wir haben Firm für diese Arbeit in entscheidenden Details erweitert. Insbesondere wurde Firm, das ursprünglich nur Programmcode beschreibt, um eine umfassende Typbeschreibung ergänzt, die unter anderem die Funktionalität einer Definitionstabelle bietet.

Zunächst führen wir die in Firm definierte Funktionalität zur Darstellung des Typsystems eines Programms ein. Wir beschreiben die für diese Arbeit essentiellen Eigenschaften dieser Darstellung. Dann erläutern wir die Darstellung des Programmcodes und die enge Verbindung der beiden Darstellungen.

4.1 Eine Typhierarchie: Darstellung der Vererbungsmechanismen

Die Zwischensprache muss die Vererbungsmechanismen des übersetzten Programms so darstellen, dass Analysen die Mechanismen einfach erfassen und erlangte Informationen präzise in der Darstellung repräsentieren können. Dazu definieren wir eine Typdarstellung die sich an Typhierarchien von Hochsprachen orientiert. Diese Darstellung repräsentiert in vielen Fällen Informationen, die durch die Hochsprache implizit festgelegt sind, explizit. So wird zum Beispiel explizit dargestellt, welche der Methoden sich gegenseitig

überschreiben. Eine Implementierung der hier definierten Typdarstellung ist in [Lin02] beschrieben.

[Tra01] definiert nur die Programmdarstellung, nicht jedoch die Darstellung von Typen und Variablen. Die ursprüngliche Verwendung von Firm durch [Tra01] in einem Übersetzer für die Programmiersprache Sather-K [**?**] stellt Typen und Variablen nur soweit dar, wie sie für die Transformation und die dort vorgestellte Optimierung nötig sind. Die Darstellung beschränkt sich auf das für Sather-K nötige.

4.1.1 Typen und Variablen

Die Typdarstellung in Firm unterscheidet zwei grundlegende Konzepte in der Darstellung: Typen und Variablen. Ein *Typ* beschreibt den erlaubten Inhalt möglicher Variablen sowie die Interpretation dieses Inhalts. Er stellt bestimmte Anforderungen an die Variable.

Eine *Variable* beschreibt eine Speicherstelle. Eine Variable gibt an, welchen Typ der Wert an dieser Speicherstelle hat, wie die Speicherstelle alloziert wird und wie auf sie zugegriffen werden kann. Variablen können aus mehreren kleineren Variablen zusammengesetzt sein. Insbesondere betrachten wir Methoden in diesem Rahmen auch als Variable, da der Methodencode Speicher belegt, adressierbar ist und durch einen Typ beschrieben werden kann. Variablen können unter anderem im Datensegment, auf dem Aufrufstapel oder auf einer Halde alloziert werden.

In den folgenden Abbildungen (4.1 ff.) stellen wir Typen in Rechtecken mit abgerundeten Kanten, Variablen in Rechtecken mit abgeschrägten Kanten dar. Beziehungen zwischen Typen und Variablen werden durch Pfeile mit durchgezogenen Strichen gekennzeichnet.

Typen

Wir führen sechs Klassen verschiedener Typen ein. Diese sind

- Basistypen
- Verbundtypen
- Verbundtypen mit Vererbung
- Referenztypen
- Reihungstypen
- Methodentypen

Basistypen beschreiben Variablen, die direkt von den Befehlen der Zielsprache gehandhabt werden können, wie zum Beispiel 16 Bit Ganzzahlen oder 64 Bit Fließkommazahlen. Basistypen legen die Abbildung ihrer Werte auf die Darstellung in der Zielsprache fest.

Verbundtypen beschreiben Ansammlungen von Variablen verschiedenen Typs. Die durch einen Verbundtyp zusammengefassten Variablen nennen wir Felder des Typs. Ein Verbundtyp der Zwischensprache listet seine Felder auf. Diese können vom Typ beliebiger Typklassen sein. Eine Instanz eines Verbundtyps ist ein Verbund.

Verbundtypen mit Vererbung sind Verbundtypen, die zusätzlich ein Reihe von Obertypen spezifizieren. Diese sind wiederum Verbundtypen mit Vererbung. Die Obertyprelation ergibt einen gerichteten Graphen, der azyklisch sein muss. Durch Invertieren des gerichteten Graphen erhalten wir die Untertyprelation. Wie sich die Felder von Verbundtypen mit Vererbung zusammensetzen, wird in Kap. 4.1.2 genauer beschrieben. Wir nennen einen Verbundtyp mit Vererbung kurz Verbundtyp, wenn aus dem Kontext klar wird, dass Verbundtypen mit Vererbung gemeint sind.

Referenztypen repräsentieren Referenzen bzw. Zeiger. Dabei wird nicht unterschieden, ob die Quellsprache Zeiger explizit kennt, wie zum Beispiel C, oder ob diese als Referenzen auf Referenzvariablen nicht explizit handhabbar sind, wie zum Beispiel in Java. Referenztypen spezifizieren den Typ der Variable, auf die die Referenz zeigt. Ist dies ein Verbundtyp mit Vererbung, kann eine Variable mit einem Referenztypen zur Laufzeit auch auf eine Variable mit einem Untertypen des spezifizierten Typs zeigen. Referenztypen legen wie Basistypen die Abbildung ihrer Werte auf die Darstellung in der Zielsprache fest.

Reihungstypen beschreiben Ansammlungen von Variablen gleichen Typs. Auch hier nennen wir die Variablen der Reihung Felder. Die Felder können in mehreren Dimensionen hierarchisch angeordnet sein. Eine Instanz eines Reihungstyps ist eine Reihung.

Methodentypen beschreiben Methoden durch Angabe der Parameter- und Ergebnistypen. Im Rahmen dieser Arbeit sind hier nur Referenz- und Basistypen zulässig. Firm erlaubt beliebige Parameter- bzw. Ergebnistypen.

Referenz-, Methoden- und Basistypen sind *atomare Typen*. Atomare Typen sind durch ihre Abbildung auf die Darstellung in der Zielsprache voll spezifiziert. Alle anderen Typen sind *zusammengesetzte Typen*. Verbundtypen sind teilweise durch die Typen ihrer Felder definiert. Zusätzlich kann die Anordnung der Felder spezifiziert werden. Dies kann, abhängig von der

Hochsprache, beim Aufbau, durch eine Optimierung, oder in einer späteren Übersetzerphase geschehen.

Variablen

Eine Variable beschreibt eine Speicherstelle. Sie referenziert den Typ, der den Wert an dieser Speicherstelle beschreibt.

Der *Besitzer* einer Variable gibt an, wo sie alloziert wird. Der Besitzer kann der globale Speicher, eine Methodenschachtel, die Halde oder ein zusammengesetzter Typ sein. In letzterem Fall spezifiziert dann eine Variable des Verbundtyps die eigentliche Allokationsstelle. Der Besitzer definiert automatisch auch den Allokationszeitpunkt der Variable. Wir fassen den globalen Speicher, die Methodenschachteln und auch die Halde als Verbundtypen auf. Damit erreichen wir für all diese Regionen eine homogene Darstellung.

Die Variabilität einer Variable beschreibt, wie auf die Variable zugegriffen werden kann. Eine Variable kann uninitialisiert, initialisiert oder konstant sein. In letzterem Fall kann auf die Variable nur lesend zugegriffen werden. Falls die Variable dynamisch alloziert wird, wird die Initialisierung implizit durch die Alloc-Operation vorgenommen (siehe Kap. 4.2). Initialisierte und konstante Variablen verfügen über Funktionalität, den Wert mit dem die Variable initialisiert werden muss, darzustellen.

Damit können Methodenfelder, bzw. Methodenvariablen, deren Darstellung für Verbundtypen essentiell ist, dargestellt werden. Methodenfelder sind konstante Variablen, die als initialen Wert die Adresse der Methode enthalten.

Für den Einsatz dieses Typmodells im Übersetzer spezifizieren Typen und Variablen noch weitere Eigenschaften, die jedoch im Rahmen dieser Arbeit nicht relevant sind.

4.1.2 Vererbung und Polymorphie

Verbundtypen mit Vererbung können Vererbung und Polymorphie explizit darstellen. Eine Variable eines Verbundtyps enthält all die Felder, die direkt als Feld dieses Typs spezifiziert sind, sowie alle Felder aller direkten Obertypen. Das heißt, Verbundtypen erben die Felder ihrer Obertypen.

Unter Polymorphie verstehen wir Polymorphie bezüglich des dynamischen Typs der Variablen, aus der das polymorphe Feld selektiert wird. Dies trifft in der Regel nur auf Methodenaufrufe zu, bei denen durch diese Variable

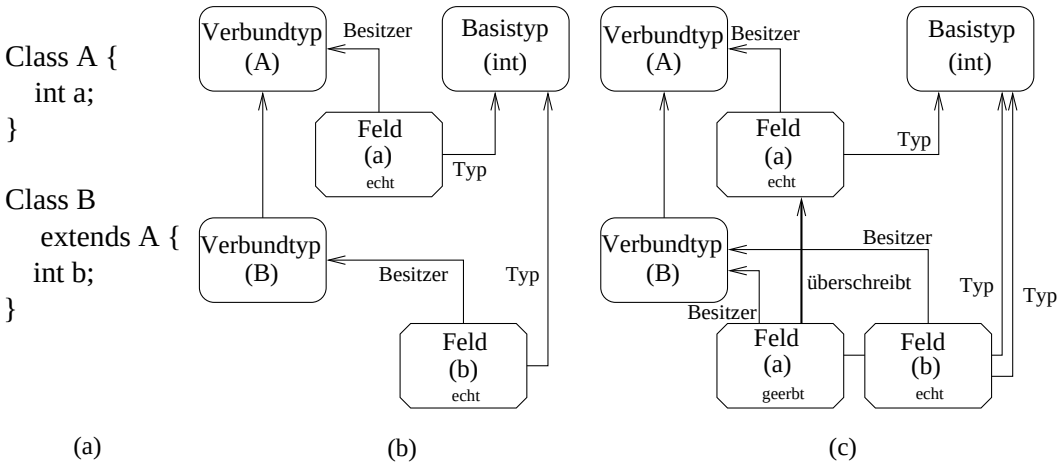

Abbildung 4.1: Darstellung der Vererbung. (a) zeigt die Typdeklaration in Pseudocode. (b) zeigt die intuitive Darstellung. Feld a wird implizit an B vererbt. (c) zeigt die explizite Repräsentation von Vererbung mittels der Überschreibt-Relation.

der Typ des impliziten `self`-Parameters bestimmt wird. Andere Polymorphie muss vor dem Aufbau der Zwischensprache aufgelöst werden. Um die Polymorphie explizit darzustellen, können Variablen, deren Besitzer ein Verbundtyp ist, in jedem direkten Obertyp des Verbundtyps ein Feld spezifizieren, das sie überschreiben. Felder eines Obertyps, die überschrieben werden, werden nicht mehr vererbt.

Damit lässt sich Vererbung auch explizit darstellen. Dazu wird in dem Verbundtypen ein Feld erzeugt, das die gleichen Eigenschaften hat wie das geerbte Feld eines Obertyps. Dieses neue Feld überschreibt das Feld des Obertyps. Abb. 4.1 zeigt ein Beispiel. Ist die Vererbung derart aufgelöst, sprechen wir von *expliziter* Vererbung, ansonsten von *impliziter* Vererbung.

Das Attribut *Ausprägung* einer Variable zeigt an, ob die Variable im Quellprogramm spezifiziert (Ausprägung *echt*), oder zur Auflösung der Vererbung erzeugt wurde (Ausprägung *geerbt*). Quellsprachen erlauben, dass eine Variable ein Feld eines transitiven Obertyps, das nicht in einem direkten Obertyp überschrieben wird, überschreibt. Dann werden auf allen Pfaden zu dem Obertyp Variablen der Ausprägung geerbt eingeführt (Abb. 4.2).

Ähnlich kann Vererbung bei Mehrfachvererbung, die auf ein gemeinsames Feld des Untertyps vererbt, bei eingeerbter Implementierung einer Schnittstelle und anderen Situationen explizit dargestellt werden.

```
Class A {
   int m() { ... };
}

Class B extends A {}
Class C extends A {}

Class D extends A, B {
   int m() { ... };
}
```

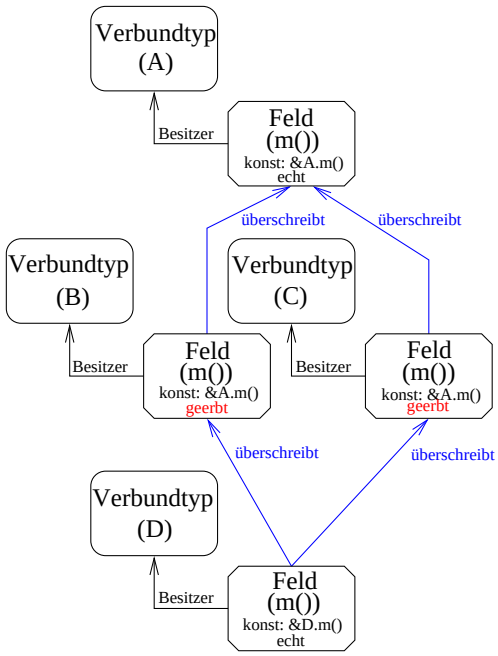

Abbildung 4.2: Darstellung entfernten Überschreibens. Typen B und C enthalten geerbte Variablen, um die Überschreibt-Relation der Methoden in A und D darzustellen. Die geerbten Variablen enthalten konstante Referenzen auf die in A definierte Methode. Die Obertyprelation wurde zwecks Leserlichkeit nicht dargestellt.

Programmiersprachen sehen in der Regel vor, dass Verbundtypen in getrennten Übersetzungseinheiten durch Vererbung erweitert werden können. Wird ein Verbundtyp übersetzt, müssen all seine Felder und Obertypen bekannt sein, da er sonst nicht vollständig spezifiziert ist. Unbekannt ist jedoch, ob es Untertypen zu diesem Typ gibt, und ob es in diesen Untertypen Felder gibt, die Felder in dem übersetzten Typ überschreiben. Programmiersprachen können jedoch Konstrukte enthalten, die weitere Untertypen oder das Überschreiben von Feldern verbieten. Dies in der Zwischensprache darzustellen ist wichtig, da für Blätter in der Untertyp- und Überschreibt-Relation bessere Optimierungsmöglichkeiten bestehen, als für innere Knoten.

Firm stellt dies mit einem ausgezeichneten Typ und einer ausgezeichnete Variable dar. Hat ein Verbundtyp diesen ausgezeichneten Typ als Untertyp, kann er außerhalb der aktuellen Übersetzungseinheit weitere Untertypen haben. Wird ein Feld von dieser ausgezeichneten Variable überschrieben, kann es außerhalb der aktuellen Übersetzungseinheit überschrieben werden. Dies setzt voraus, dass der Besitzertyp den ausgezeichneten Typ als Untertyp hat.

4.2 Darstellung der Programmoperationen: Explizite Feldzugriffe und Typisierung des Programmcodes

Firm stellt Programmcode in SSA-Form als einen gerichteten Datenflussgraphen dar. Abhängigkeiten über dem Speicher werden durch eine funktionale Modellierung des Speichers explizit dargestellt.

4.2.1 Programmcode in Firm als Datenflussgraph

Knoten in diesem Graph sind Operationen des Programms. Jedem Operand dieser Operationen entspricht eine eingehende Datenflusskante. Die Operationen haben genau ein Ergebnis. Ausgehende Kanten stellen den Datenfluss dieses Ergebnisses dar. Das Ergebnis kann ein Tupel mehrerer Werte sein. Eine spezielle Operation selektiert einzelne Werte aus einem Tupel. Operationen mit Seiteneffekten wie der Methodenaufruf Call oder die Speicheroperationen Load und Store haben den funktionalen Speicher als Operanden und den veränderten funktionalen Speicher als Ergebnis, eventuell als Element eines Tupels.

Zusätzliche Attribute an den Operationen spezifizieren deren Verhalten genauer. Der Übersetzer nutzt diese Attribute wenn er hochsprachnahe Operationen zu zielsprachnäheren transformiert. Die Attribute können aber auch Zusatzinformationen für Analysen enthalten. Firm wird im Detail in [TLB99] beschrieben. Die Grundstruktur der Darstellung richtet sich nach [Tra01], wir haben jedoch verschiedene Operationen verallgemeinert und normalisiert, um in Firm Programme quellsprachunabhängig darstellen zu können.

In den Abbildungen stellen wir Operationen als Rechtecke dar, Datenflusskanten bzw. Operanden durch Pfeile mit durchgezogenen Linien. Pfeile mit gestrichelten Linien stellen Attribute dar. Die Operation zur Selektion eines einzelnen Wertes aus einem Tupel stellen wir als kleinen Kasten am unteren Rand der Operation dar. Kanten die Datenabhängigkeiten über dem Speicher modellieren, lassen wir weg, wenn sie für das Verständnis der Abbildung unnötig sind.

4.2.2 Die Sel-Operation

Die Operation Sel zur Feldselektion wurde bereits in [Tra01] eingeführt. Die Sel-Operation selektiert ein einzelnes Feld aus einem zusammengesetzten Typ. Da das Datensegment und Stapelschachteln mit Verbundtypen dargestellt werden, kann die Sel-Operation auch aus diesen Variablen selektieren. Ein Attribut der Sel-Operation verweist auf die Repräsentation des zu selektierenden Felds in der Typdarstellung. Die Operanden der Sel-Operation sind eine Referenz, ein Speicherwert und gegebenenfalls Reihungsindizes.

Mit der hier definierten Typdarstellung können wir die Semantik wie folgt konkretisieren. Die Referenz referenziert eine Instanz des Besitzertyps des zu selektierenden Felds. Falls dieser Typ ein Verbundtyp mit Vererbung ist, kann die Referenz dynamisch auch eine Instanz eines Untertyps referenzieren. Wird in diesem Untertyp das Feld überschrieben, wird dieses überschreibende Feld selektiert. Der durch den Speicherwert repräsentierte Speicher muss diese Instanz enthalten. Reihungsindizes spezifizieren die zu selektierende Variable wenn der Besitzertyp ein Reihungstyp ist. Abb. 4.3 illustriert die Semantik der Sel-Operation.

Das Ergebnis der Sel-Operation ist eine Referenz auf das selektierte Feld.

Die Sel-Operation versteckt den Adressierungsmechanismus der Felder, insbesondere den polymorpher Methoden. Durch die Sel-Operation wird die Darstellung kompakt. Datenflussanalysen müssen nicht eine komplizierte

```
Class A {
   void m() { ... };
}

Class B extends A {
   void m() { ... };
}

A a;
...
a.m();
```

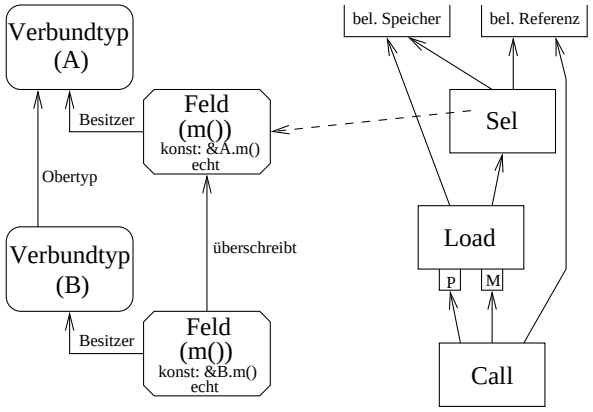

Abbildung 4.3: Polymorphe Feldselektion mit der Sel-Operation. Referenziert der Wert des als *beliebige Referenz* bezeichneten Knotens eine Instanz des Typs A, zeigt das Ergebnis der Sel-Operation auf eine Speicherzelle, die eine Referenz auf A.m() enthält. Ist es eine Instanz des Typs B, zeigt das Ergebnis auf eine Speicherzelle die eine Referenz auf B.m() enthält.

Adressberechnung analysieren, um zu erkennen, auf was genau zugegriffen wird. Die Sel-Operation stellt alle für die Analyse notwendigen Information zur Verfügung. Die Darstellung des Zugriffs auf ein Feld mit der Sel-Operation öffnet verschiedene Optimierungsmöglichkeiten, da der Zugriffsmechanismus leicht angepasst werden kann.

Eine spätere Phase im Übersetzer ersetzt die Sel-Operation durch expliziten Adressberechnungscode.

4.2.3 Hochsprachtypisierung der Programmcodedarstellung

Alle Operationen in Firm sind mit dem Maschinentyp, mit dem ihr Ergebnis dargestellt werden soll, annotiert. Firm, wie in [Tra01] spezifiziert, sieht jedoch keine vollständige Typisierung mit Hochsprachtypen bzw. den Typen des Firm Typsystems vor. Lediglich Operationen, deren Semantik durch einen Typ konkretisiert wird, wie Sel, Alloc oder Call, verweisen in die Typdarstellung.

Eine Zwischensprache benötigt keine Hochsprachtypisierung, wenn vor dem Aufbau der Zwischensprache Typpassfehler beseitigt wurden. Typpassfehler treten einerseits bei Operationen auf Basistypen auf. Solche Passfehler

werden durch explizite Typanpassungen, die das Bitmuster der Maschinentypen manipulieren, behoben. Dazu steht in Firm die Operation Conv zur Verfügung, die zum Beispiel einen 64 Bit Fließkommawert unter Genauigkeitsverlusten in einen 16 Bit Ganzzahlwert umwandelt.

Andererseits garantieren bestimmte Hochsprachen Kompatibilität von Verbundtypen. Die Sprache C garantiert Kompatibilität indem sie die Anordnung der Verbundtypen fest definiert (wenn auch abhängig von der Zielmaschine). Die Anordnung kann nicht optimiert werden. Daher kann diese fest in der Zwischendarstellung dargestellt werden, ohne Optimierungsspielraum zu verlieren. Objektorientierte Sprachen garantieren Kompatibilität innerhalb der Vererbungshierarchie. Dadurch können komplexe Umwandlungsoperationen notwendig werden, die zum Beispiel Referenzwerte ändern. Um diese Umwandlungen auf Zielsprachniveau zu formulieren, ist in der Regel eine genaue Kenntnis der Speicheranordnung der Objekte nötig. Lässt die Anordnung der Objekte Optimierungsspielraum offen, möchte man die Umwandlungen erst nach der Optimierung konkretisieren. Die Zwischensprache muss Funktionalität bieten, die Umwandlungen auf Hochsprachebene darzustellen.

Die Cast-Operation

Dazu führen wir die Cast-Operation ein. Cast ist eine unäre Operation die als Attribut auf einen Typ verweist. Sie wandelt den Hochsprachtyp des Wertes ihres Operanden um. Hat dieser Wert einen statischen Typ, der direkt kompatibel ist, ändert die Operation den Wert nicht. Ansonsten führt sie eine implizite Umwandlung durch.

Eine spätere Phase im Übersetzer muss alle Cast-Operationen durch den expliziten Umwandlungsalgorithmus ersetzen. Dazu ist es nötig, auch den statischen Typ des Operanden zu kennen. Um dies zu erreichen, erweitern wir Firm so, dass für jede Operation, die einen echten Wert darstellt, der statische Typ bekannt ist.

Dazu ist es ausreichend, alle Quellen von Werten mit expliziten Verweisen in die Typdarstellung zu versehen.

Typisierte Quellen von Werten

Quellen von Werten sind die Operationen Start, Const, Alloc, Load und Call. Start und Call verweisen bereits auf die entsprechenden Methodentypen, die

Tabelle 4.1: Firm-Operationen mit Verweisen in die Typbeschreibung

Operation	Referenzierte Typinformation
Alloc	Typ der allozierten Variable
SymConst$_{AddrEnt}$	Adresse einer statisch allozierten Variable
Sel	Selektierte Variable
Call	Signaturtyp der aufgerufenen Methode
Call	Liste aller Methodenvariablen die hier aufgerufen werden können (Aufrufgraph)
Cast	Zieltyp der Typanpassung
Const	Typ der Konstanten
SymConst$_{Size}$	Typ dessen Größe die Konstante darstellt
SymConst$_{TypeTag}$	Typ dessen Kennzeichen die Konstante darstellt

die Parameter bzw. Ergebnistypen spezifizieren. Die Alloc-Operation verweist bereits auf den Typ der allozierten Variable. Der Typ des Ergebnisses ist der entsprechende Referenztyp. Ist die Quellsprache streng typisiert, kann man an einer Load-Operation den Typ des geladenen Wertes aus dem Referenztypen des Referenzoperanden der Operation ableiten. Die Const-Operation haben wir um ein Attribut, das auf die Typdarstellung verweist, erweitert.

Nun lassen sich die Typen aller anderen Werte über die Datenflusskanten direkt ableiten. Hat eine Operation Operanden mit verschiedenen Typen, spezifiziert die Semantik der Operation den Ergebnistyp.

4.3 Ergänzende Optimierungen der Typhierarchie

In der Typdarstellung von Firm lassen sich Ergebnisse von Analysen über der Vererbungshierarchie explizit darstellen. Dadurch kann die Effizienz der Pufferoptimierung und auch anderer Optimierungen gesteigert werden, ohne dass diese explizit auf Informationen der Analysen zugreifen müssen. Dabei wird die Typdarstellung verfeinert, so dass sich unabhängige Verwendungen von Feldern und Instanzen von Typen in der Programmdarstellung ablesen lassen. Im Folgenden gehen wir davon aus, dass die implizite Vererbung aufgelöst ist.

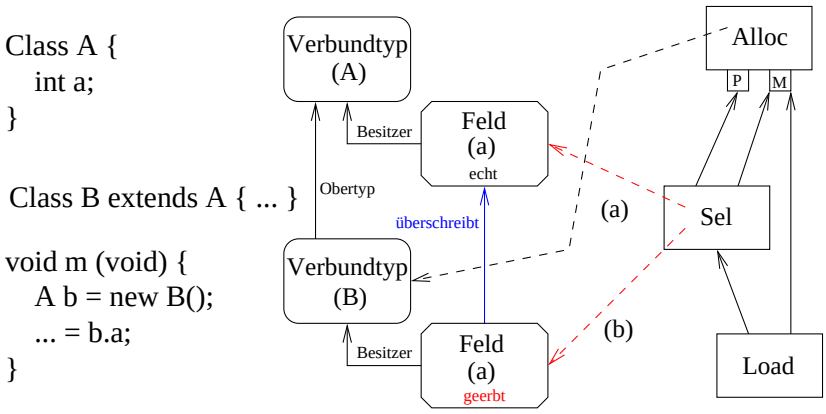

Abbildung 4.4: Präzisierung der Programmdarstellung mit geerbten Feldern. Wird Kante (a) durch (b) ersetzt, drückt dies aus, das an dieser Stelle immer ein Feld von Verbund B selektiert wird.

4.3.1 Feldzugriffe konkretisieren

Durch die Darstellung geerbter Felder können Feldzugriffe präziser dargestellt werden. Dazu muss eine Analyse den Referenztyp des Operandens einer Sel-Operation bestimmen. An diesem kann man den referenzierten Verbundtypen ablesen. Ist dies ein Untertyp des durch die Quellprogrammtypisierung vorgegebenen Typs, kann die Sel-Operation konkretisiert werden. Dazu sucht man im Untertyp das Feld, das das durch die Sel-Operation als Attribut referenzierte Feld überschreibt. Dieses Feld setzt man als neues Attribut der Sel-Operation ein. Abb. 4.4 zeigt ein Beispiel.

Weitere Analysen können nun Informationen über diese Sel-Operation gezielt für den Untertypen sammeln. Die Information wird nicht mehr mit dem Obertyp, den sie gar nicht betreffen kann, in Zusammenhang gebracht. Optimierungen können den Untertyp für diese Sel-Operation optimieren, unabhängig von Anforderungen des Obertyps. Umgekehrt kann die Sel-Operation für den Untertypen optimiert werden. Selektiert die Sel-Operation eine Methode, und wird das eingesetzte Feld nicht weiter überschrieben, so kann nach dieser Optimierung ein polymorpher Methodenaufruf entfernt werden.

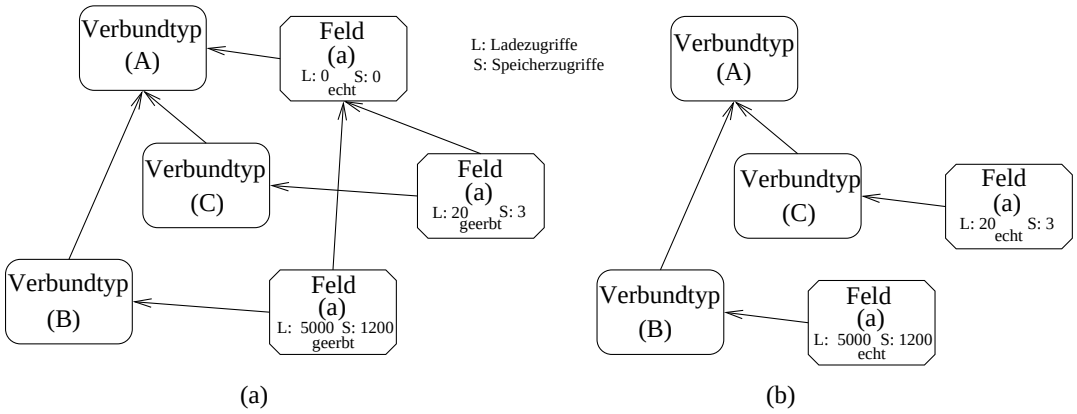

Abbildung 4.5: Wird das Feld a nie in dem Verbund A zugegriffen, so kann es aus diesem Verbund entfernt werden. Möchte man im Typgraphen (a) den Verbund B so optimieren, dass Feld a effizienter zugegriffen wird, müssen auch Verbunde A und C optimiert werden. Für den Typgraphen (b) kann Verbund B isoliert optimiert werden.

4.3.2 Felder vereinzeln

Obertypen faktorisieren häufig Gemeinsamkeiten eigentlich verschiedener Typen aus. Instanzen des Obertypen werden nicht verwendet. Definiert solch ein Obertyp ein Feld, das nur in den Untertypen verwendet wird, ist es nicht nötig, für dieses Feld Typkompatibilität zwischen den Untertypen zu erzwingen. Abb. 4.5 zeigt ein Beispiel dieses Szenarios. Diese Situation in der Darstellung kann nur durch die Voroptimierung aus Kap. 4.3.1 entstehen, die alle Zugriffe auf A.a entfernt hat.

Wird eine oberste Variable, das heißt, eine Variable die kein Feld in einem Obertyp überschreibt, von keiner Sel-Operation referenziert, so kann sie aus der Typbeschreibung entfernt werden. Die Variablen, die die entfernte Variable überschrieben haben, müssen nun als *echt* markiert werden. Sie können nun unabhängig voneinander optimiert werden, insbesondere wenn einige als pufferkritisch, die anderen als unkritisch klassifiziert wurden.

Kann eine oberste Variable wegen weniger Sel-Operationen nicht entfernt werden, können diese Sel-Operationen durch explizite Typtests vermieden werden. Abhängig von dem Typtest kann eine Sel-Operationen auf eine überschreibende Variable ausgeführt werden. Jeder Zugriff kann dann passend optimiert werden. Die übersetzte Hochsprache sollte dazu bereits eine dynamische Typisierung vorsehen, da die Einführung einer solchen durch den

Übersetzer in der Regel nicht sinnvoll ist. Die Kosten der expliziten Tests müssen sich durch nachfolgende Optimierungen amortisieren.

4.3.3 Typhierarchie vereinfachen

Wird ein Typ nie zu einem seiner Obertypen umgewandelt, kann diese Obertyprelation mit allen parallelen Überschreibt-Relationen gelöscht werden. Vorher muss Vererbung explizit aufgelöst sein. Dadurch wird explizit dargestellt, dass diese Typen nicht typkompatibel sein müssen. Man gewinnt neue Freiheiten für das Layout. Dies schafft mehr Freiheitsgrade für die Optimierung des Typs, vermeidet Zusatzaufwand um die Typkompatibilität zu erhalten und gibt weiteren Typanalysen mehr Möglichkeiten.

Eine Analyse kann eventuell feststellen, dass es unabhängige Mengen von Verbunden des selben Typs gibt. Das heißt Verbunde einer Menge erreichen nie die gleichen Programmstellen wie die Verbunde einer anderen Menge. Die Analyse kann für jede dieser Mengen eigene Typen einführen. Da die Mengen unabhängig sind, gibt es keine Typumwandlungen zwischen Elementen der Mengen. Nun kann eine Optimierung für die neuen Typen getrennte Optimierungsstrategien entwickeln.

Eine Analyse kann für innere Verbunde feststellen, dass deren Verwendung teilweise disjunkt zur Verwendung der Untertypen ist. Dies kann man durch einen neuen Untertyp des inneren Verbunds darstellen. Dieser Untertyp erbt alle Felder des inneren Verbunds. Alle Allokationen des inneren Verbunds werden in Allokationen des neuen Verbunds geändert. Alle Programmstellen, die nur von dem inneren Verbund erreicht werden, nicht aber von seinen Untertypen, werden auf die Verwendung der Felder des neuen Verbunds geändert. Der innere Verbund dient nur noch der Darstellung der Gemeinsamkeiten seiner Untertypen.

4.4 Eignung der Darstellung

Mit den oben beschriebenen Eigenschaften stellt Firm eine besonders geeignete Basis für unsere Optimierungen dar. Es konserviert alle wichtigen Informationen aus der Quellsprache in der Struktur der Darstellung (Typen, Vererbung, Überschreiben, Einschränkungen der Feldanordnung, Feldzugriffe, Typumwandlungen, Signatur von Aufrufen). Diese und zusätzliche Analyseinformationen (Aufrufrelation, Steuerfluss, vollständige Typisierung

des Programmcodes, Datenflussinformation) lassen sich direkt aus der Struktur der Darstellung ablesen. Da Firm diese Informationen bis kurz vor der Assemblerausgabe korrekt hält, kann die Pufferoptimierung zu einem sehr späten Zeitpunkt in der Übersetzung stattfinden. Dadurch können vorher Standardoptimierungen ausgeführt werden, die die Effizienz der Pufferoptimierung steigern, da sie die eigentlichen Strukturen des zu optimierenden Programms besser darstellen (Entfernen toten und unerreichbaren Codes, polymorphe Aufrufe auflösen, Typhierarchie vereinfachen, offener Einbau). Die Ergebnisse dieser Optimierungen werden direkt in der Zwischensprache dargestellt, sodass die Pufferoptimierung keine Schnittstelle zu den Standardoptimierungen benötigt.

Die knappe Darstellung, die zusammengehörige Informationen direkt verbindet, steigert die Effizienz der Pufferoptimierung weiter. Ergebnisse der Analysephase der Pufferoptimierung können direkt an die Konstrukte der Zwischendarstellung annotiert werden. Die konservierten Informationen aus der Quellsprache stellen alle für die Transformationen durch die Optimierung notwendigen Informationen zur Verfügung (Feldselektion, Typumwandlungen, Methodensignaturen beim Aufruf, Überschreibtbeziehung).

Wie die einzelnen Eigenschaften der Zwischendarstellung verwendet werden zeigen die nächsten beiden Kapitel, die die Analyse und Optimierung beschreiben.

Kapitel 5

Auffinden pufferkritischer Verbunde

Dieses Kapitel stellt Analysen zum Auffinden pufferkritischer Verbunde und Datenstrukturen vor. Diese Analysen sollen Verbunde mit den in Kap. 2.6 diskutierten Pufferproblemen identifizieren. Die Analyseergebnisse dienen als Grundlage für die Optimierung der Verbunde.

Dabei sollen Verbunde erkannt werden, von denen zur Laufzeit so viele Instanzen entstehen können, dass diese nicht alle gleichzeitig im Pufferspeicher gehalten werden können. Weiter soll die Analyse pufferkritische Felder identifizieren. Dazu verwenden wir Analysen des Steuer- und Datenflusses und der Typdarstellung. Diese Analysen arbeiten auf dem gesamten, dem Übersetzer verfügbaren Programm.

In der in Kap. 4 gegebenen Zwischendarstellung sind alle Feldzugriffsoperationen und Allokationsoperationen im übersetzten Programm bekannt. Können wir die Ausführungshäufigkeiten dieser Operationen angeben, absolut oder relativ, können wir durch Akkumulation daraus die Anzahl der Allokationen und Feldzugriffe ableiten. Im Folgenden schlagen wir eine Analyse vor, die die Ausführungshäufigkeit einzelner Operationen abschätzt. Diese setzt sich aus einer Analyse der Ausführungshäufigkeiten von Operationen in Methoden und einer Analyse der Ausführungshäufigkeiten von Methoden zusammen. Dann zeigen wir, wie wir Felder identifizieren, die zur Iteration über Datenstrukturen dienen. Schließlich stellen wir eine Heuristik zur Erkennung pufferkritischer Verbunde und Datenstrukturen vor.

5.1 Analyse der Ausführungshäufigkeiten

Ein Grundblock ist eine maximale Sequenz von Operationen, die ausgeführt werden, wenn nur eine der Operationen aus dem Block ausgeführt wird [Muc97]. Die Ausführungshäufigkeit einer Operation ist also genau die Ausführungshäufigkeit ihres Grundblocks. Die Ausführungshäufigkeit von Grundblöcken hängt vom umgebenden Steuerfluss ab.

5.1.1 Zusammenflüsse und Verzweigungen

Die Ausführungshäufigkeit eines Grundblocks ist die Summe der Ausführungshäufigkeiten der Kanten zu seinen Vorgängern. Der Grundblock, der den Eintritt in eine Methode darstellt, hat keinen Vorgänger und wird initial mit Eins bewertet. Andere Grundblöcke ohne Vorgänger sind toter Code und werden mit Null bewertet.

Hat ein Grundblock genau eine Ausgangskante, so wird diese mit der Ausführungshäufigkeit des Grundblocks bewertet. Ist ein Grundblock eine Verzweigung, so hat er mehrere Ausgangskanten, auf die sich die Ausführungshäufigkeit des Blocks aufteilt. Das Verhältnis, in dem die Ausführungshäufigkeit aufgeteilt wird, hängt von der Bedingung für den bedingten Sprung der Verzweigung ab. Dieses kann nur für einen konkreten Programmlauf mit einer Eingabe genau angegeben werden. Statisch nehmen wir eine feste Verteilung an, zum Beispiel das alle Nachfolger der Verzweigung gleich häufig angesprungen werden.

Vergleiche auf Null dienen häufig als Abbruchbedingung oder als Test auf einen ungewünschten Zustand. Man könnte also annehmen, dass Vergleiche mit Null in der Regel fehlschlagen. Wir konnten dies durch Messungen (Kap. 7.2.1) nicht belegen und führen keine gesonderte Bewertung von Nullvergleichen ein.

Hat ein Grundblock mehrere Vorgänger, darunter eine Verzweigung, nennen wir die Steuerflusskante zwischen der Verzweigung und dem Zusammenfluss kritischen Steuerfluss. Wir schließen kritischen Steuerfluss durch Normalisierung aus. Dazu platzieren wir auf jeder Steuerflusskante zwischen einem Grundblock mit mehreren Vorgängern und einer Verzweigung einen neuen, leeren Block, wie in Abb. 5.1.

Diese Normalisierung wird auch für andere Optimierungen, wie Elimination Partieller Redundanzen, benötigt. Der entstandene leere Block kann durch eine einfache Optimierung wieder entfernt werden. Durch diese Nor-

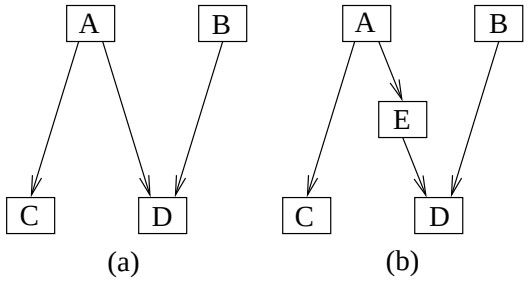

Abbildung 5.1: Block A ist eine Verzweigung, Block D ein Zusammenfluss. Durch Einfügen des leeren Blocks E vermeiden wir den kritischen Steuerfluss zwischen A und D.

malisierung muss für einen Block entweder der Fall Verzweigung oder der Fall Zusammenfluss berücksichtigt werden.

5.1.2 Schleifen

Die Werte für Zusammenflüsse und Verzweigungen, die eine Schleifenrückwärtskante enthalten, lassen sich mit dieser Bewertung nicht angeben, da eine zyklische Abhängigkeit der Ausführungshäufigkeiten besteht.

Zusätzlich möchten wir Blöcke in Schleifen mit einer höheren Ausführungshäufigkeit bewerten als die Blöcke vor und nach der Schleife. Daher zerlegen wir den Steuerflussgraphen mit Intervallanalyse in azyklische und zyklische Regionen. Wir berechnen für jede Region in der Zerlegung die Ausführungshäufigkeiten.

Wir vergeben die Ausführungshäufigkeiten entlang dem durch die Intervallanalyse konstruierten Baum. Für eine azyklische Region gehen wir nach dem in Kap. 5.1.1 eingeführten Schema vor. In einer zyklischen Region mit genau einem Schleifenkopf weisen wir dem Schleifenkopf die an der Region annotierte Ausführungshäufigkeit zu, nachdem wir diese mit der geschätzten Iterationszahl der Schleife multipliziert haben (Beispiel Abb. 5.2). Bei reduziblen Programmen ist der Schleifenkopf der Dominator aller Blöcke in dieser Region und bestimmt daher die Ausführungshäufigkeiten.

Für irreduziblen Steuerfluss legen wir künstlich einen Schleifenkopf fest, und behandeln die Region dann wie eine reduzible Schleife. Dadurch wird das Betreten der irreduziblen Region über verschiedene Blöcke dem künstlichen Schleifenkopf zugeschlagen und es ergeben sich Ungenauigkeiten. Wir nehmen jedoch an, dass das Betreten der Schleife relativ zu den Iterationen

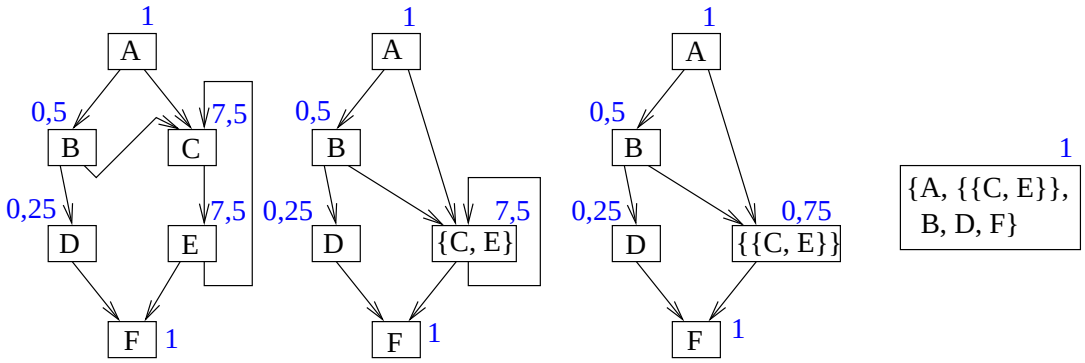

Abbildung 5.2: Ausführungshäufigkeiten mit Intervallanalyse. Schleifen werden mit Faktor 10, Verzweigungen gleichverteilt bewertet. Die Abfolge von links nach rechts zeigt die Dekomposition der Intervallanalyse. Die Ausführungshäufigkeiten werden von rechts nach links vergeben.

der Schleife wenig Bedeutung hat und vernachlässigen dies. In den Testprogrammen, mit denen wir in Kap. 7 Messungen durchführen, kommt kein irreduzibler Steuerfluss vor.

Berechnung der Intervallzerlegung

Für die Zerlegung des Steuerflusses in Intervalle berechnen wir zunächst starke Zusammenhangskomponenten und stellen diese in einem Schleifenbaum dar. Dazu verwenden wir den Algorithmus aus [Tra01]. Im Gegensatz zu [Tra01] betrachten wir jedoch nur die Blockoperationen der Zwischensprache für den Schleifenbaum, das heißt wir berechnen den Baum nur über dem Steuerfluss. Da der Steuerflussgraph deutlich kleiner ist als der Datenflussgraph, ist die Berechnung des Schleifenbaumes über dem Steuerfluss schneller. Nach der Berechnung des Schleifenbaumes sind alle Rückwärtskanten als solche markiert.

Abb. 5.4 zeigt einen Steuerflussgraphen und den dazugehörigen Schleifenbaum, Abb. 5.5 die Schleifenbäume zu den Intervallbeispielen in Abb. 5.2 und Abb. 5.3.

Die Knoten der Intervallzerlegung sind die Blockknoten und alle Schleifenknoten. Die Kanten der Intervallzerlegung leiten wir von den Kanten zwischen den Blöcken ab. Dabei bewerten wir eine Kante $b1 \rightarrow b2$ zwischen zwei Blöcken $b1$ und $b2$, die in Schleifen $s1$ und $s2$ liegen, wie folgt:

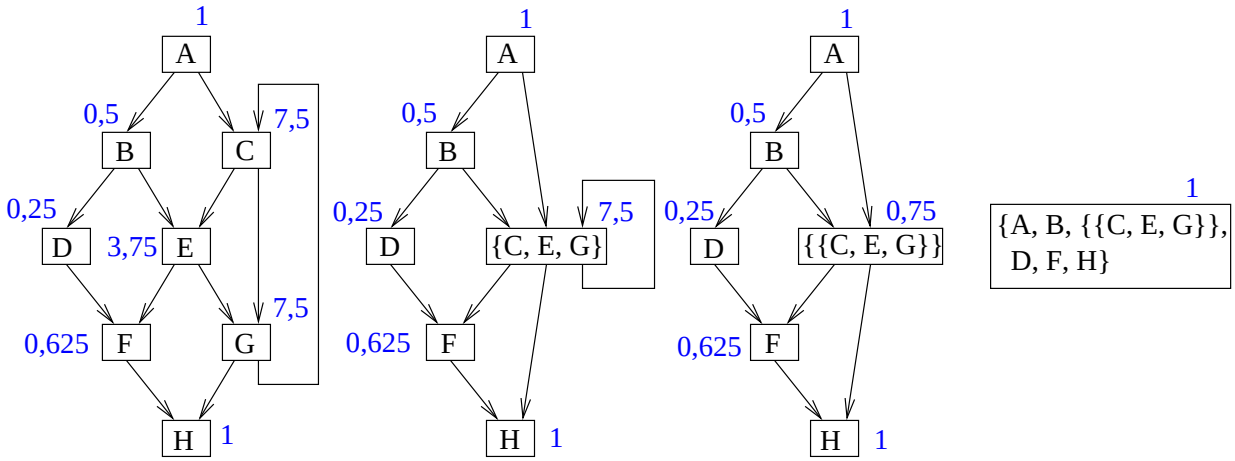

Abbildung 5.3: Ausführungshäufigkeiten mit Intervallanalyse für irreduziblen Steuerfluss. Die Vergabe der Ausführungshäufigkeiten unterscheidet nicht, ob die Schleife über Block C oder E betreten wurde.

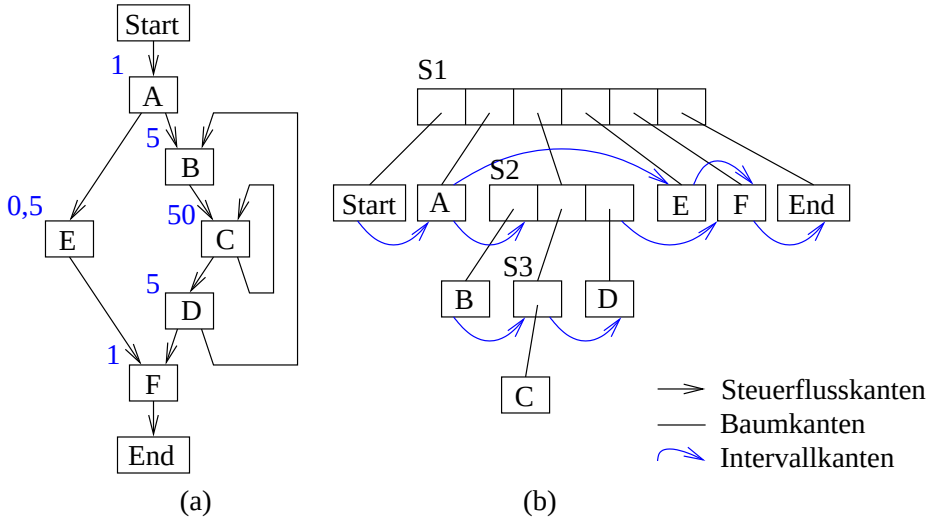

Abbildung 5.4: Schleifenbaum mit geschachtelten Schleifen. In weiteren Bildern stellen wir Blätter im Schleifenbaum der Übersichtlichkeit zu liebe nicht mehr als eigene Knoten dar.

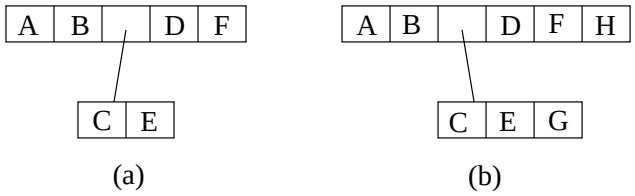

Abbildung 5.5: Schleifenbäume zu Intervallbeispielen in Abb. 5.2: (a), Abb. 5.3: (b).

- Ist $b1 \rightarrow b2$ eine Rückwärtskante, streichen wir die Kante. (Abb. 5.4, Kante $D \rightarrow B$.)
- Liegen die Blöcke $b1$ und $b2$ im gleichen Schleifenknoten ($s1 == s2$), übernehmen wir die Kante in den Intervallgraphen. (Abb. 5.4, Kante $A \rightarrow E$.)
- Ist $s2$ eine innere Schleife von $s1$, suchen wir die Schleife $s3$, die direkt in $s1$ enthalten ist, und $s2$ enthält. Wir fügen die Kante $b1 \rightarrow s3$ hinzu. (Abb. 5.4, $s1 = S2, s2 = S3, s3 = S3$, Kante $B \rightarrow C$ wird $B \rightarrow S3$.)
- Ist $s1$ eine innere Schleife von $s2$, suchen wir die Schleife $s3$, die direkt in $s2$ enthalten ist, und $s1$ enthält. Wir fügen die Kante $s3 \rightarrow b2$ hinzu. (Abb. 5.4, $s1 = S2, s2 = S1, s3 = S2$, Kante $D \rightarrow F$ wird $S2 \rightarrow F$.)
- Ist weder $s1$ in $s2$, noch $s2$ in $s1$ enthalten, suchen wir die tiefste Schleife $s3$, die $s1$ und $s2$ enthält. Ist $s4$ die Schleife, die $s1$ enthält und ihrerseits direkt in $s3$ enthalten ist, und genauso $s5$ die Schleife, die $s2$ enthält und ihrerseits direkt in $s3$ enthalten ist, fügen wir die Kante $s4 \rightarrow s5$ hinzu.

5.1.3 Ausnahmen

Moderne Programmiersprachen verwenden Ausnahme-Konstrukte um Programmfehler elegant abfangen zu können. Da diese Ereignisse selten sind, bewerten wir Steuerflussverzweigungen aufgrund von Ausnahmen mit einer sehr geringen Wahrscheinlichkeit.

Firm stellt den Steuerfluss, der durch Ausnahmen entsteht, mittels erweiterten Grundblöcken dar [Rie04]. Ein erweiterter Grundblock hat eine

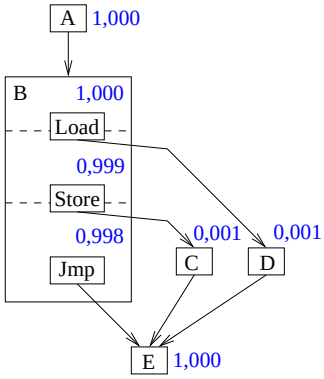

Abbildung 5.6: Block B ist ein erweiterter Grundblock. Wir nehmen an, dass Ausnahmen mit der Wahrscheinlichkeit 1/1000 auftreten. Die ausnahmebehafteten Operationen teilen den Block in Bereiche, die mit verschiedener Ausführungshäufigkeit bewertet werden müssten. Wir wollen jedoch eine einheitliche Bewertung des Blocks. Bewerten wir B mit dem Wert nach der letzten Ausnahme, 0,998, und die Ausnahmeverzweigungen wie gehabt, ergibt sich für Block E die neutrale Bewertung 1,000.

Einsprungstelle, aber mehrere Aussprungstellen [Muc97]. Innere Aussprungstellen sind in Firm ausschließlich Sprünge aufgrund eintretender Ausnahmen. Der eigentliche Steuerfluss beendet den erweiterten Grundblock. In dieser Darstellung können wir die geringere Ausführungshäufigkeit von Operationen nach einer nicht aufgetretenen Ausnahme nicht an den Block annotieren, um dies von der Ausführungshäufigkeit von Operationen vor der Ausnahme zu unterscheiden. Für die Bewertung der Operationen ist dies von geringer Bedeutung, da wir das Eintreten von Ausnahmen als unwahrscheinlich annehmen. Dies werden wir in Kap. 7.2.1 belegen. Die Summe der Ausführungshäufigkeiten nach einem erweiterten Grundblock sollte jedoch mit der Summe davor übereinstimmen, siehe Abb. 5.6. Daher bewerten wir den gesamten erweiterten Grundblock mit der Ausführungshäufigkeit nach der letzten nicht eingetretenen Ausnahme.

5.1.4 Methoden und Methodenaufrufe

Bisher haben wir die Ausführungshäufigkeiten von Blöcken in einer Methode abgeschätzt. Für die Bewertung eines Blocks in einem Programm ist die Ausführungshäufigkeit der Methode des Blocks jedoch ebenso wichtig.

Verbinden wir jeden Aufruf mit dem Beginn aufgerufener Methoden, und das Ende der aufgerufenen Methoden mit der Stelle nach dem Aufruf, erhalten wir einen interprozeduralen Steuerflussgraphen. Auf diesem könnten wir die gleiche Heuristik anwenden, wie für intraprozeduralen Steuerfluss. Da Aufrufe in der Regel keiner syntaktischen Beschränkung durch die Programmiersprache unterliegen, kann dieser Graph aber nicht hierarchisch zerlegt werden wie ein intraprozeduraler Steuerflussgraph. Nicht reduzible Regionen sind die Regel. Zyklen in diesem Graphen entsprechen nicht Rekursionen im Programm, und es besteht kein direkter Zusammenhang zwischen Methodeneintritt und Methodenende.

Daher verwenden wir zur Bestimmung der Ausführungshäufigkeit von Methoden den Aufrufgraphen. Dieser gibt an, aus welchen Methoden welche anderen Methoden aufgerufen werden können.

Grundsätzlich ergibt sich die Ausführungshäufigkeit eines Blocks aus der Ausführungshäufigkeit des Blocks im intraprozeduralen Steuerfluss multipliziert mit der Ausführungshäufigkeit der Methode. Es genügt also, zusätzlich zu den Ausführungshäufigkeiten im intraprozeduralen Steuerfluss die Ausführungshäufigkeiten der Methoden zu berechnen. Diese ergeben sich jedoch wiederum aus der Summe der Ausführungshäufigkeiten aller Aufrufe der Methode, die ihrerseits von einer Block-Ausführungshäufigkeit abhängen.

Die Knoten des Aufrufgraphen sind die bei der Übersetzung bekannten Methoden. Zwischen Methoden $m1$ und $m2$ existiert eine Kante $m1 \rightarrow m2$ wenn ein Aufruf in $m1$ möglicherweise die Methode $m2$ aufruft. Methoden, für die nur die Signatur, nicht aber die Implementierung bekannt ist, können nicht Ursprung einer Kante sein. Hier ist der Aufrufgraph ungenau. Der Aufrufgraph enthält einen Knoten, der für alle nicht bekannten Methoden steht. Sind nicht alle Aufrufer einer Methode bekannt, so ist dieser Knoten ein Aufrufer dieser Methode.

Berechnung des Aufrufgraphen

Um den Aufrufgraphen zu berechnen, müssen wir für jede Aufrufstelle die Methoden, die hier aufgerufen werden können, angeben. Dazu müssen wir die Adressausdrücke der Methodenaufrufe analysieren. Für diese gibt es drei grundlegend verschiedene Möglichkeiten.

Der Adressausdruck kann eine symbolische Konstante sein. In diesem Fall geht der Aufruf an eine eindeutige, bekannte Methode.

Der Adressausdruck kann eine polymorphe Methodenselektion sein. Der

Ausdruck spezifiziert die höchste Methode aus der Vererbungshierarchie, die aufgerufen werden kann. Wir nehmen in erster Näherung an, dass alle Methoden, die diese Methode überschreiben, aufgerufen werden können.

Für Methodenselektionen, die direkt von dem Referenzergebnis einer Allokation selektieren, kennen wir die aufgerufene Methode. In unserer graphbasierten SSA-Darstellung können wir direkt den Typ der Variable, von der die Methode selektiert wird, ablesen. Ist die Variable a in Abb. 4.4 eine Methode, und wird der Verbund B weiter überschrieben, können wir aufgrund der vorangehenden Allokation schließen, dass hier die Implementierung von a aus Verbund A, die nach B vererbt wird, aufgerufen wird. Weiter sehen wir in unserer Typdarstellung sofort, ob die selektierte Methode ein Blatt im Vererbungsbaum ist, also nicht weiter überschrieben werden kann. Solche Methodenselektionen haben wir bereits durch konstante Aufrufe ersetzt. Tabelle 7.3 zeigt die Bedeutung dieser Optimierung.

Steht uns an der Methodenselektion weitere Information über den dynamischen Typ zur Verfügung, schränken wir die Menge der aufgerufenen Methoden mit dieser Information zusätzlich ein. Solche Information kann aus einer Typ- oder Haldenanalysen stammen. Hier profitieren wir auch von Voroptimierungen nach Kap. 4.3.1.

Letztendlich kann der Adressausdruck eine beliebige Adresse sein, die zum Beispiel aus einem Load Knoten stammt. In diesem Falle müssen wir annehmen, dass hier alle Methoden aufgerufen werden. Unter den Voraussetzungen dieser Arbeit kommt dies jedoch nicht vor.

Bewertung der Ausführungshäufigkeit von Methoden auf dem Aufrufgraph

Der Aufrufgraph unterscheidet sich vom Steuerflussgraphen grundlegend. Der Steuerflussgraph enthält zwei disjunkte Knoten, die für Ausführungsbeginn und Ausführungsende stehen. Dazwischen folgt die Ausführung den Graphkanten. Der Knoten für das Ausführungsende ist das einzige Blatt im Graphen (bzw. in einer Zusammenhangskomponente des Graphen). Im Aufrufgraphen beginnt die Ausführung jedoch in einem dedizierten Knoten und steigt entlang der Graphkanten ab. Spätestens wenn sie ein Blatt im Aufrufgraph erreicht, kehrt sie um und folgt dem bisher beschrittenen Pfad im Aufrufgraph in umgekehrter Richtung, bis sie an einem beliebigen Knoten wieder umkehrt und den Pfad wieder verlängert. Terminiert das Programm, ist der Pfad in der Regel komplett abgebaut.

Methoden, die außerhalb der Übersetzungseinheit sichtbar sind, sind mögliche Einsprungpunkte in den übersetzten Programmteil. Gehen wir von Ganzprogrammübersetzung aus, ist dies genau die Methode die nach Definition der übersetzten Programmiersprache als initialer Aufrufpunkt vorgesehen ist. Wir bewerten diese Methode mit der Ausführungshäufigkeit 1.

Um die Ausführungshäufigkeit einer Methode $m2$ zu bewerten, müssen alle Vorgänger von $m2$ im Aufrufgraph bewertet sein. Dann bewerten wir $m2$ mit der Summe der Bewertungen aller eingehenden Kanten, wobei eine Kante $m1 \rightarrow m2$ wie folgt bewertet wird:

Die Kante $m1 \rightarrow m2$ abstrahiert n Aufrufe $call_0 \ldots call_{n-1}$ in $m1$ an $m2$. Die Ausführungshäufigkeit des Aufrufs $call_i$, $freq(call_i)$, ist die Ausführungshäufigkeit des Blocks, in dem der Aufruf ist, multipliziert mit der Ausführungshäufigkeit der Methode, in der dieser Block ist. Ein Aufruf entspricht im interprozeduralen Steuerfluss einer bedingten Verzweigung. Von $call_i$ können l_{call_i} verschiedene Methoden aufgerufen werden. Wie bei Verzweigungen teilen wir die Ausführungshäufigkeit des Aufrufknotens in erster Näherung gleichermaßen auf alle aufgerufenen Methoden auf. Die Ausführungshäufigkeit der Kante $m1 \rightarrow m2$ ergibt sich also zu

$$freq(m1 \rightarrow m2) = \sum_{i=0}^{n-1} (freq(block(call_i))/l_{call_i} * freq(m1)).$$

Die Ausführungshäufigkeit $freq(m2)$ der Methode $m2$ mit o Aufrufern ergibt sich also zu

$$freq(m2) = \sum_{i=0}^{o-1} (freq(mi \rightarrow m2).$$

Wie der Steuerflussgraph kann der Aufrufgraph Zyklen enthalten. In Zyklen ist es nicht möglich, für alle Vorgänger die Ausführungshäufigkeit zu berechnen. Wir kennen Rückwärtskanten in Zyklen und berücksichtigen diese bei der Berechnung nicht. Wir berechnen die Ausführungshäufigkeiten für Methoden in einer abgewandelten Tiefensuche, bei der wir einen Knoten erst dann besuchen, wenn wir alle seine Vorgänger besucht haben.

Durch das Auslassen der Rückwärtskanten werden Rekursionen unterbewertet. Daher bewerten wir Rekursionen gesondert.

Die Zyklen im Aufrufgraph stellen mögliche Rekursionen dar. Sie sind eine obere Schranke der Rekursionen im Programm. Da vom tatsächlichen Steuerfluss in der Methode abstrahiert wird, kann nicht erkannt werden, wenn

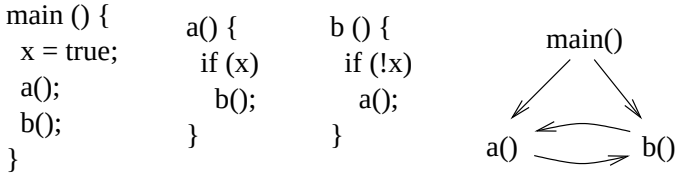

Abbildung 5.7: Scheinbare Rekursion. a() und b() rufen sich nie zyklisch auf da sie selbst die Bedingung x nicht verändern. Trotzdem ergibt sich ein Zyklus im Aufrufgraph.

zwei Aufrufe sich gegenseitig ausschließen. Dies zeigt das Beispiel in Abb. 5.7. Dies passiert insbesondere bei polymorphen Aufrufen. Ein Beispiel ist hier die Java Methode `toString()`. Diese wird zur Ausgabe von Objekten verwendet. Werden die Felder eines Verbunds wieder mit dieser Methode ausgegeben, kann durch die Polymorphie eine scheinbare Rekursion entstehen.

Weiter werden die Rekursionen durch die Ungenauigkeiten bei der Abschätzung der Aufrufrelation überschätzt.

Wir erkennen zunächst die Zyklen im Aufrufgraph durch Berechnung starker Zusammenhangskomponenten. Wir markieren eine Kante in jedem Zyklus als Rückwärtskante bzw. rekursiven Aufruf. Die Knoten, die zu diesem Zyklus beitragen, stellen eine potentielle Rekursion dar. Wir merken uns zu jedem Zyklus die zugehörigen Knoten in einem Schleifenbaum. Es gibt keine Pfade aus einem Zyklus zurück zu Knoten in einem Zyklus, der auf dem Pfad vom Einsprungspunkt in das Program zu diesem Zyklus durchquert wurde. Geschachtelte Rekursionen sind voneinander unabhängige Zyklen im Aufrufgraph, wie Abb. 5.8 zeigt.

Liegt der Einsprungspunkt zum Programm in einem Zyklus, so fügen wir dem Aufrufgraph einen künstlichen Vorgänger hinzu. Der Einsprungspunkt bekommt die Rekursionstiefe 0 zugewiesen.

Wir berechnen mit einer Tiefensuche die Anzahl der Zyklen, die wir auf einem Suchpfad gefunden haben. Auch hier besuchen wir einen Knoten erst, wenn wir alle seine Vorgämger besucht haben. Wir weisen einem Knoten die maximale Anzahl Zyklen zu, die wir auf einem Weg zu diesem Knoten durchquert haben. Rückwärtskanten berücksichtigen wir dabei wieder nicht.

Wir erhalten zwei Bewertungen für eine Methode: Die Ausführungshäufigkeit sowie die Rekursionstiefe. Die gesamte Ausführhäufigkeit $freq_{inter}$ eines Blocks b in einem Programm bewerten wir dann mit folgender Formel:

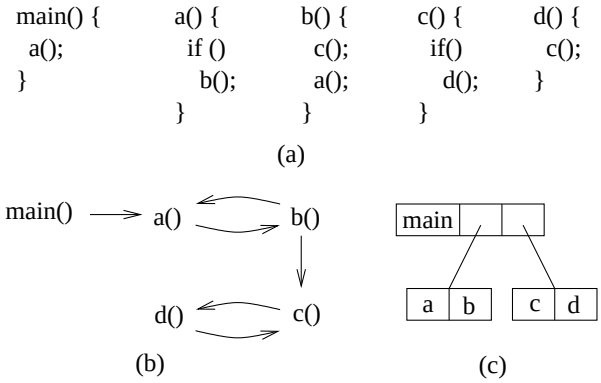

Abbildung 5.8: Geschachtelte Rekursion. (a) Beispielprogramm, (b) Aufrufgraph, (c) Schleifenbaum

$$freq_{inter}(b) = freq_{intra}(b) * (freq(m(b)) + c^{rek\_depth(m(b))}),$$

wobei c eine frei gewählte Konstante ist. Durch Wahl von c lässt sich die Gewichtung von interprozeduraler Ausführhäufigkeit und Rekursionstiefe einstellen.

5.2 Erkennen von Kantenfeldern in Datenstrukturen

Wir teilen Felder dynamischer Datenstrukturen in zwei Klassen. Solche, die die eigentlichen Nutzdaten enthalten, und solche, die zur Navigation in der Datenstruktur verwendet werden. Letztere nennen wir *Kantenfelder*. Kantenfelder werden prinzipiell öfter zugegriffen, da man ein Kantenfeld laden muss, um daraufhin ein Datenfeld zu laden, nicht jedoch umgekehrt. Insbesondere sind solche Kantenfelder interessant, auf die rekursiv zugegriffen wird. Das heißt, dass aus dem Verbund, auf den ein Kantenfeld zeigt, das nächste Kantenfeld mit dem gleichen Zugriff geladen wird. Solch einen Zugriff nennen wir rekursiven Zugriff ([Cah02]). Abb. 5.9 zeigt Programmstücke mit rekursiven Zugriffen.

Alle Felder, die Referenzen auf andere Verbunde enthalten, sind Kantenfelder. Rekursive Zugriffe können jedoch nicht mit allen Kantenfeldern

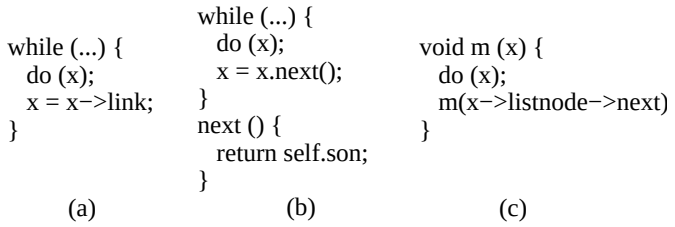

Abbildung 5.9: Rekursive Zugriffe

erreicht werden. Kantenfelder müssen im Typgraphen einen Zyklus bilden, damit mit ihnen rekursive Zugriffe möglich sind. Enthalte zum Beispiel ein Verbund T_1 ein Kantenfeld das Verbunde vom Typ T_2 referenziert. Über dieses Kantenfeld sind nur dann rekursive Zugriffe möglich, wenn man über Kantenfelder wieder von T_2 nach T_1 gelangen kann, wenn also zum Beispiel T_2 ein Kantenfeld enthält, dass T_1 referenziert. Solche Kantenfelder nennen wir rekursive Kantenfelder.

Im Folgenden beschreiben wir, wie wir rekursive Kantenfelder erkennen.

5.2.1 Erkennen von rekursiven Kantenfeldern

Rekursive Kantenfelder sind Kantenfelder, die Zyklen in einem Graphen bilden, der Verbundtypen als Knoten und Kantenfelder als Kanten enthält. Genauer ist dies der Graph, der durch die Feldrelation der Zwischensprache gegeben ist.

Bei Verbundtypen muss zusätzlich die Untertyprelation berücksichtigt werden. Referenziert ein Kantenfeld einen Obertypen, von dem es keinen Pfad zum Besitzer des Kantenfeldes gibt, kann dieses Feld durch Vererbung trotzdem rekursiv sein. Dies ist der Fall, wenn ein Pfad von einem Untertypen des referenzierten Typen zum Besitzer des Kantenfeldes existiert, da jede Referenz auf einen Obertypen auch einen Untertypen referenzieren kann.

Wir konstruieren zunächst einen Graphen indem wir die Typdarstellung der Zwischensprache traversieren und alle Verbundtypen in diesen Graphen aufnehmen. Wir untersuchen die Typen der Felder jedes aufgenommenen Verbundtypen T_1. Ist der Feldtyp ein Referenztyp, finden wir den Typ, auf den die Referenz zeigt, auch über mehrere Indirektionen, ansonsten betrachten wir direkt den Feldtyp. Ist dieser ein Verbundtyp T_2 fügen wir eine Kante $T_1 \rightarrow T_2$ in der Graphen ein. Ist T_2 ein Verbundtyp mit Vererbung fügen wir zusätzlich Kanten von T_2 zu allen Untertypen von T_2 ein.

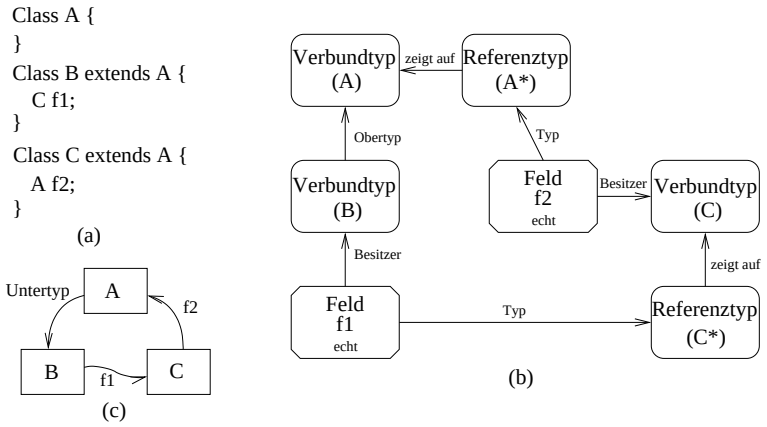

Abbildung 5.10: Graph zum Finden rekursiver Kantenfelder. (b) zeigt die Typdarstellung in der Zwischensprache für das Programm in (a). (c) zeigt den daraus abgeleiteten Graphen. Felder f1 und f2 sind Kanten in einer starken Zusammenhangskomponente, sind also Kantenfelder. Die dritte Kante ist die Untertypbeziehung.

Nun berechnen wir starke Zusammenhangskomponenten auf diesem Graphen. Wir haben uns für jede Kante gemerkt, zu welchem Feld sie korrespondiert. Nach dem Berechnen der Zusammenhangskomponenten markieren wir jede Kante in einer starken Zusammenhangskomponente als rekursives Kantenfeld.

5.3 Identifikation pufferkritischer Verbunde

In diesem Kapitel möchten wir Verbunde und Datenstrukturen auf der Ebene der Typdarstellung bezüglich ihrer möglichen Pufferprobleme (Siehe Kap. 2.6) charakterisieren. Dazu ziehen wir die Ausführungshäufigkeiten aus Kap. 5.1 heran. Diese sind jedoch bisher an Grundblöcken in der Codedarstellung annotiert. Die folgenden Kapitel zeigen, wie diese Information in die Typdarstellung übernommen wird, und dort die gewünschten Eigenschaften kondensiert werden.

5.3.1 Akkumulation der Ausführungshäufigkeiten auf Typebene

Wir erhalten die Summe aller Zugriffe auf ein Feld durch Summierung der Ausführhäufigkeit aller Zugriffsoperationen auf dieses Feld. Gleiches gilt für Allokationen einer Struktur.

Dazu invertieren wir in einem einfachen Lauf über die Codedarstellung die Beziehungen zwischen den Operationen und der Typdarstellung. Wir annotieren an jedem Typ bzw. an jeder Variablen die Operationen, die auf sie verweisen.

Die Sel-Operation stellt im Allgemeinen eine Adressberechnung dar, nicht jedoch einen Speicherzugriff. Berechnet eine Sel-Operation eine Feldadresse, ist sie Operand einer Lade- oder Speicheroperation, die den eigentlichen Speicherzugriff darstellt. Daher sammeln wir zusätzlich alle Lade- und Speicheroperationen (Load und Store) auf und annotieren diese an den Variablen, die wir aus dem Adressausdruck kennen. Selektiert eine Sel-Operation eine Methode, wird diese als aufzurufende Methode an eine Call-Operation weitergegeben. Die Adressberechnung, die durch die Operation dargestellt wird, benötigt einen Speicherzugriff auf eine Variable, um die Adresse der Sprungtabelle zu laden, und einen weiteren um die Methodenadresse zu laden. Um die Häufigkeiten der Zugriffe auf die Adresse der Sprungtabelle abzuschätzen, müssen wir also die Ausführungshäufigkeit der Sel-Operationen, die Methoden selektieren, berücksichtigen.

5.3.2 Bewertung einzelner Variablen und Typen

Anzahl Instanzen eines Typs

Wir schätzen die Anzahl dynamisch allozierter Instanzen ab, indem wir die Ausführungshäufigkeiten aller Allokationsoperationen für jede Typ aufsummieren.

Diese selbstständig allozierten Instanzen sind nicht alle Instanzen eines Typs. Es können statisch allozierte Variablen eines Typs existieren. Diese können wir zur Übersetzungszeit einfach zählen, da sie alle bekannt sind. Weiter enthalten Methodenschachteln Variablen. Da mit jedem Methodenaufruf eine Methodenschachtel angelegt wird, ist die Anzahl der Aufrufe einer Methode eine obere Schranke für die Anzahl dieser Variablen. Schließlich können Variablen in anderen, zusammengesetzten Variablen enthalten

sein. Solche Variablen werden mit der Allokationshäufigkeit des Typs der umgebenden Variable bewertet. Da sich Variablen nicht rekursiv enthalten können, ist diese Relation nicht zyklisch und wir können immer eine Bewertung angeben.

Allerdings profitieren diese Variablen nicht von den Pufferoptimierungen, da wir die Platzierung von Variablen, die nicht auf der Halde alloziert werden, nicht steuern können. Daher berücksichtigen wir diese nicht.

Anzahl Zugriffe

Die Anzahl der Zugriffe auf ein Feld schätzen wir ab, indem wir die Ausführungshäufigkeiten aller Lade- und Speicheroperationen summieren. Für Methodenvariablen summieren wir die Ausführungshäufigkeiten der Sel-Operationen, da diese den Zugriff über die Sprungtabelle abstrahiert, der Speicherzugriffe bedingt. Wir nennen diese Summen auch die *Zugriffshäufigkeiten* der Felder.

5.3.3 Bewertung von Variablen und Typen in einer Vererbungshierarchie

In der Darstellung von Vererbungshierarchien in Firm (siehe Kap. 4.1.1 und 4.1.2) spezifiziert ein Typ eine Variable nur teilweise. Die Spezifikation des Typs wird durch Einerben von Feldern aus Obertypen eines Typs vervollständigt. Wir gehen im Folgenden davon aus, dass implizite Vererbung aufgelöst wurde, und Analysen Verwendungen von geerbten Variablen im Programmcode eingeführt haben, wie dies in Kap. 4.3.1 beschrieben ist.

Anzahl Allokationen eines Feldes

Wir kennen aus Kap. 5.3.2 die Anzahl Allokationen für jeden Typ in der Vererbungshierarchie. Ein Feld, das in einem Typ T spezifiziert ist, ist auch in allen Untertypen UT von T enthalten. Das heißt, Zugriffe auf das Feld umfassen auch Zugriffe auf das Feld in den Untertypen. Eine komplett getrennte Bewertung eines Feldes als Teil von T oder als Teil von UT ist nicht möglich. Wir können das Feld auch nicht abhängig von dem Typ, mit dem es alloziert wurde (Besitzertyp), optimieren, da wir dann den Zugriff auf das Feld dynamisch vom Besitzertyp abhängig machen würden. Dies bedeutet einen nicht zu rechtfertigenden Zusatzaufwand.

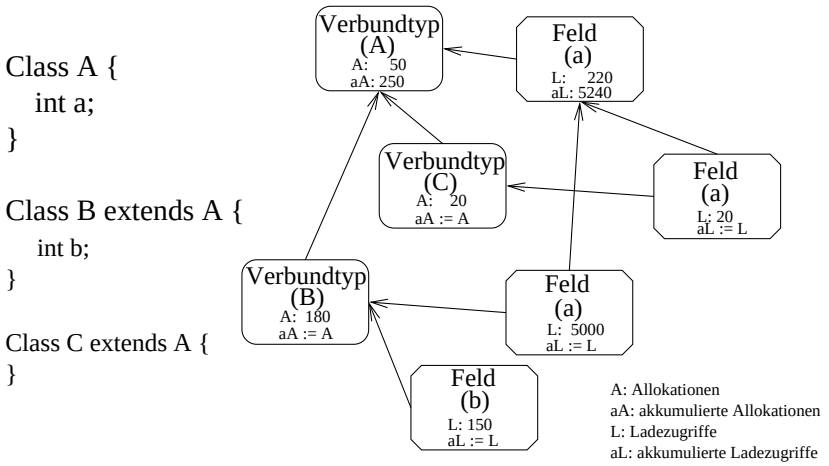

Abbildung 5.11: Individuelle und Akkumulierte Allokations- und Ladehäufigkeiten. Die Pfeile entsprechen den in Abb. 4.4 eingeführten.

Daher ziehen wir für die Bewertung eines Feldes die gesamte Anzahl seiner Vorkommen heran, die akkumulierte Anzahl Vorkommen. Diese berechnet sich aus der Anzahl an Vorkommen in allen Untertypen und ist für alle Felder eines Typs gleich. Ein Feld kommt in jeder Instanz einmal vor, daher entspricht die Anzahl der Vorkommen der Anzahl Allokationen. Da die Vererbungshierarchie azyklisch ist, kann die akkumulierte Allokationshäufigkeit für einen Typ aus der akkumulierten Allokationshäufigkeit seiner Untertypen berechnet werden.

Wir erhalten diese Größe für Typen, indem wir die Vererbungshierarchie von den Blättern, den am meisten spezifizierten Typen, zu den Wurzeln, das heißt den allgemeinsten Typen, ablaufen, und unterwegs die Untertypen in einer Menge aufsammeln. Für jeden Typ summieren wir dann die Allokationen aller Typen in dieser Menge. Wir verwenden eine Menge, um zu vermeiden, dass wir Untertypen bei Mehrfachvererbung doppelt zählen.

Genauso akkumulieren wir die Anzahl der Zugriffe auf Felder. Dies müssen wir jedoch für jedes Feld getrennt vornehmen. Abb. 5.11 zeigt exemplarisch die Berechnung der akkumulierten Allokations- und Ladehäufigkeiten.

Weiter berechnen wir die Speicherhäufigkeiten, Häufigkeiten rekursiver Zugriffe und Häufigkeiten der Zugriffe auf die Sprungtabelle. Für diese summieren wir zusätzlich die Zugriffshäufigkeit über alle Felder, die in der

Sprungtabelle platziert werden. Dies sind neben Methoden alle konstanten Felder, die in Abhängigkeit des Besitzertyps zugegriffen werden müssen.

5.3.4 Auswahl pufferkritischer Verbunde und Felder

Mit der bisher beschriebenen Analyse haben wir eine Abschätzung, wie viele Verbunde eines Typs existieren, und wie oft auf die Felder der Verbunde zugegriffen wird. Ziel dieses Kapitels ist es, anhand dieser Werte Typen zu identifizieren, die überdurchschnittlich viele Fehler verursachen könnten, und daher pufferkritisch sind.

Dazu sortieren wir Typen und Felder anhand der Allokations- und Zugriffshäufigkeiten und klassifizieren alle oberhalb eines Schwellenwertes als pufferkritisch.

Schwellenwerte

Wir können keine absoluten Schwellenwerte verwenden, da die erhobenen Daten auf Annahmen beruhen. Stattdessen suchen wir Werte, die relativ zu den anderen groß sind. Wir verwenden verschiedene Verfahren, einen Schwellenwert festzulegen. Abb. 5.12 illustriert die im Folgenden diskutierten Ansätze.

Wir betrachten eine Menge $W = \{w_0, ..., w_{n-1}\},\ w_i > w_{i+1}$ sortierter absoluter Werte und suchen eine Teilmenge $C \subset W$ pufferkritischer Werte deren Elemente $c_0, ...c_{m-1},\ n > m,\ c_i = w_i$ oberhalb einer Schwelle s liegen.

Wir können den Schwellenwert s relativ zum größten Wert festlegen: $s = w_0 * f_1,\ 0 < f_1 < 1$. Dann werden alle Werte die größer als, zum Beispiel, die Hälfte des größten Werts sind, als oberhalb des Schwellenwertes eingestuft. Sind dann jedoch alle Werte sehr ähnlich, werden alle Werte gleich eingestuft. Ist umgekehrt der größte Wert ein Ausreißer, kann es sein das nur dieser eine Wert die Schwelle überschreitet.

Alternativ können wir den Schwellenwert so festlegen, dass alle großen Werte die einen gewissen Anteil der Gesamtheit ausmachen, oberhalb des Schwellenwertes liegen. Dann suchen für $0 < f_2 < 1$ ein $m < n$ mit

$$s = w_m \quad \wedge \quad \sum_{i=0}^{m-1}(c_i \in C) \leq \sum_{i=0}^{n-1}(w_i \in W) * f_2 < \sum_{i=0}^{m-1}(c_i \in C) + c_m.$$

Diesen Schwellenwert berechnen wir durch Iteration auf der sortierten Liste der Werte. Hier kann es passieren, dass der Schwellenwert zwei Werte

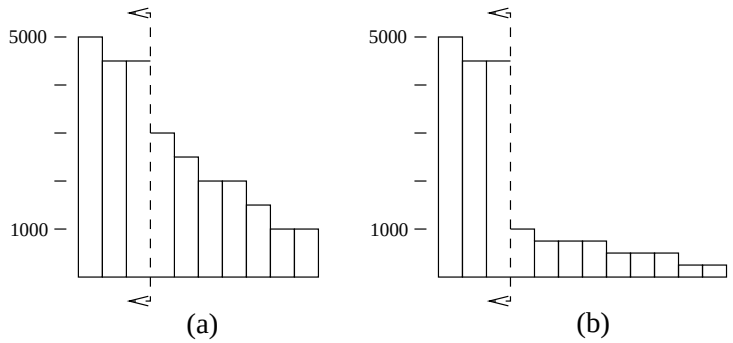

Abbildung 5.12: Die Werte links der gestrichelten Linie werden als ‚groß' eingestuft. (a): Die ersten drei Werte (zusammen 14000) sind mehr als die Hälfte ($f_2 = 0,5$) der Gesamtsumme (27000). (b): Zwischen dem dritten und vierten Wert ist ein großer Sprung ($f_3 = 4$): $w_3 > 4 * w_4$.

trennt, die sich sehr ähnlich sind. Wenn es sehr viele kleine Werte gibt, können auch viele kleine Werte die Schwelle überschreiten.

Dies vermeidet eine Strategie, die Unterschiede zwischen benachbarten Werten in der Sortierung berücksichtigt. Bei dieser wird der Schwellenwert dort festgelegt, wo erstmals ein großer Sprung in der Sortierung zu verzeichnen ist. D.h. $s = w_m$, wobei m der kleinste Wert ist, für den gilt: $w_m * f_3 > w_{m+1},\ 0 < f_3$. Findet sich kein solches m, kann man f_3 iterativ kleiner wählen.

Erkennung pufferkritischer Felder

Hier versuchen wir einzelne Felder in Verbundtypen zu identifizieren, die durch häufige Zugriffe für Probleme in der Pufferleistung führen können. Diese Felder haben eine hohe zeitliche Wiederverwendung. Dabei suchen wir zunächst eine Bewertung anhand der Zugriffshäufigkeiten, unabhängig von der Einstufung als rekursive Kantenfelder.

Interessant ist hier vor allem, wie oft auf ein Feld einer Instanz des Besitzertypen zugegriffen wird. Diesen Wert nennen wir *Instanz-Zugriffshäufigkeit* im Gegensatz zu der Zugriffshäufigkeit, die sich auf die Summe aller Zugriffe auf das Feld über alle Instanzen bezieht.

Um eine Pufferoptimierung für ein Feld zu rechtfertigen, sollte die Instanz-Zugriffshäufigkeit deutlich größer als eins sein, da der erste Zugriff ein obligatorischer Fehler ist. Eine Feldvariable repräsentiert alle Laufzeitinstanzen

eines Feldes. Daher berechnet sich die Instanz-Zugriffshäufigkeit auf den ersten Blick aus der Zugriffshäufigkeit geteilt durch die Allokationshäufigkeit des Besitzertyps. In einer Vererbungshierarchie subsumiert die Zugriffshäufigkeit jedoch auch Zugriffe auf Untertypen, die Allokationshäufigkeit ist aber typspezifisch. Daher verwenden wir akkumulierte Werte. Aus dem gleichen Grund dürfen wir vererbte Felder nicht direkt bewerten. Wir geben ihnen die gleiche Bewertung wie dem ursprünglichen Feld, das ganz oben in der Überschreibt-Relation steht. Gibt es aufgrund von Mehrfachvererbung mehrere solche, bilden wir einen Mittelwert. Genauer betrachten wir als oberste Felder die obersten Felder, deren nicht-akkumulierte Zugriffshäufigkeit größer Null ist.

Die Instanz-Zugriffshäufigkeit ist ein gutes Maß, wenn die Zugriffe auf alle Instanzen der Feldvariable gleich verteilt sind. Das ist nicht immer gegeben, so werden zum Beispiel die Sohnfelder der Wurzel eines Baumes sehr oft zugegriffen, die Sohnfelder von Blättern hingegen sehr selten. In solchen Fällen sollten die wenigen, pufferkritischen Felder bevorzugt behandelt werden, bzw. die Variablen, in denen sie vorkommen. Eine Optimierung des Typs als ganzes ist dann nicht sinnvoll. Eine derart gezielte Optimierung ist also auf Typebene nicht möglich. Zusätzlich können wir einzelne Instanzen nicht unterscheiden, dies würde einen nicht zu rechtfertigenden Analyseaufwand bedeuten. Durch Voroptimierungen können jedoch verschiedene Präzisionen der Typdarstellung erreicht werden, wie in Kap. 4.3 diskutiert. Deren Ergebnisse sind dann jedoch direkt im Typgraphen dargestellt.

Anhand der Instanz-Zugriffshäufigkeit bewerten wir die Felder mit zwei Heuristiken. Einerseits suchen wir unter allen Feldern die mit der höchsten Instanz-Zugriffshäufigkeit. Andererseits bewerten wir die Felder eines Typs. Hier betrachten wir alle Felder als pufferkritisch, deren Instanz-Zugriffshäufigkeit mindestens 10% ($f_1 = 0,1$) der maximalen Instanz-Zugriffshäufigkeit aller Felder des Typs beträgt.

Stabile Felder

Weiter schätzen wir ab, ob sich eine Datenstruktur schnell ändert, oder relativ stabil bleibt. Ändert sich der Inhalt von Kantenfeldern nach dem ersten Schreiben nicht oder sehr selten, besteht eine feste Beziehung zwischen zwei Verbunden. Wird das Kantenfeld oft gelesen, besteht vermutlich hohe räumliche Wiederverwendung zwischen den Verbunden.

Wir bezeichnen ein Feld, dessen Inhalt sich im Vergleich zu den Zugrif-

fen selten ändert, als *stabil*. Dazu vergleichen wir die Ladehäufigkeiten und Speicherhäufigkeiten.

Wir bezeichnen ein Feld als stabil, wenn #Ladehäufigkeit $\gg$ #Speicherhäufigkeit und #Speicherhäufigkeit $\cong$ #Allokationshäufigkeit. Auch hier verwenden wir akkumulierte Zahlen und bewerten nur oberste Felder.

Erkennung pufferkritischer Typen

Um Typen, deren Instanzen zu pufferkritischen Datenstrukturen beitragen können, zu erkennen, verwenden wir verschiedene Maße.

Wir verwenden die geschätzte Anzahl Allokationen eines Typs als Maß für die Größe der Datenstruktur, die daraus zusammengesetzt werden kann. Wir berücksichtigen die Typen, die zusammen 99% ($f_2 = 0,99$) aller Allokationen ausmachen. Dies ist in der Regel nur ein geringer Teil aller Typen, da von den meisten Typen nur wenige oder gar keine Instanzen alloziert werden.

Weiter schätzen wir die Anzahl Zugriffe pro Instanz des Typs ab. Dazu summieren wir über alle Felder des Typs deren Instanz-Zugriffshäufigkeit, und teilen dies durch die voraussichtliche Größe des Typs. Als Schwellenwert verwenden wir die durchschnittliche Zugriffshäufigkeit berechnet für die Gesamtheit aller Typen.

Kapitel 6

Optimierung pufferkritischer Verbunde

Bisher haben wir gezeigt, wie wir pufferkritische Verbunde mit statischer Analyse erkennen können. Dieses Kapitel zeigt, wie mit diesen Informationen Optimierungen der Datenanordnung vorgenommen werden können. Dabei bauen wir auf den Arbeiten aus Kap. 3.1.2 auf.

Zunächst präsentieren wir Typanhäufen. Mit dieser Technik optimieren wir die Pufferleistung. Typanhäufen erhöht die räumliche Wiederverwendung zwischen Verbunden durch gezielte Allokation. Bei unveränderten Algorithmen erhöht sich dadurch auch die räumliche Lokalität, was zu weniger Pufferfehlern führt. Dann stellen wir mehrere Verbesserungen von Verbundteilen vor, das die räumliche Lokalität innerhalb von Verbunden erhöht. Wir präsentieren Varianten von Verbundteilen, die die von der Hochsprache geforderte Typkompatibilität respektieren, aber auch solche die explizite Typumwandlungen nötig machen.

6.1 Typanhäufen

Typanhäufen [GTZ98] ist, im Gegensatz zu Anhäufen [CHL99], eine Anordnungsoptimierung, die wenig Zusatzkosten bedingt. Dafür kann Anhäufen spezifische Eigenschaften optimierter Programme besser ausnutzen. Dies macht die automatische Anwendung von Anhäufen jedoch schwierig, da diese Eigenschaften erkannt werden müssen. Wendet man Anhäufen aufgrund ungenauer Analysedaten an, können die Kosten die Gewinne leicht übersteigen.

Typanhäufen und Anhäufen nutzen beide Kenntnisse über pufferkritische Kantenfelder aus. Hat ein Verbund ein pufferkritisches Kantenfeld, werden der Verbund und der durch das Kantenfeld referenzierte Verbund häufig nacheinander zugegriffen. Sie haben einen geringen Zugriffsabstand. Wenn die Verbunde in der gleichen Pufferzeile liegen, haben sie räumliche Wiederverwendung. Da der Zugriffsabstand gering ist, ist die Wahrscheinlichkeit hoch, dass auf die Pufferzeile ein weiteres mal zugegriffen wird, bevor sie verdrängt wird. Anhäufen platziert derartige Verbunde in einer gemeinsamen Pufferzeile.

Anhäufen kann direkt in ein Programm eingebaut werden. Dazu muss bei der Allokation des ersten Verbunds so Speicherplatz für den referenzierten Verbund reserviert werden, dass sie in eine gemeinsame Pufferzeile fallen. Die Allokation des referenzierten Verbunds muss dann diesen vorreservierten Platz verwenden. Dies erfordert Laufzeittests um festzustellen, ob schon Speicher für einen Verbund vorgesehen ist. Weiter verursacht Anhäufen Speicherverschnitt, da nicht immer vorhergesehen werden kann, ob in das Kantenfeld eine Referenz auf einen neu allozierten Verbund geschrieben wird. Wird eine Referenz auf einen bestehenden Verbund oder gar keine Referenz in das Feld geschrieben, bleibt der vorreservierte Speicher unbenutzt. Um solche Situationen zu vermeiden, muss die Optimierung das Programm sehr genau kennen. Daher wurde Anhäufen so bisher nur in halbautomatischen Werkzeugen eingesetzt.

Zusätzlich ist Anhäufen nur für stabile Felder sinnvoll. Die Beziehung zwischen den angehäuften Verbunden muss mindestens so lange stabil bleiben, bis sich der Zusatzaufwand amortisiert hat. Wird das Kantenfeld neu geschrieben, ist das Programm zwar immer noch korrekt, es besteht jedoch keine Lokalität mehr zwischen den sich referenzierenden Verbunden. Ist bekannt, wann sich eine Datenstruktur stark ändert, kann die Datenstruktur danach umkopiert werden, um wieder Lokalität herzustellen. Dies ist jedoch mit sehr hohen Kosten verbunden, da neben dem eigentlichen Kopieren auch alle Referenzen auf die Verbunde angepasst werden müssen.

Alternativ kann Anhäufen mit einem kopierenden Speicherbereiniger erreicht werden. Kennt dieser relevante Kantenfelder, kann er die Objekte, die sich zum Zeitpunkt des Kopierens referenzieren, zusammen in eine Pufferzeile platzieren. Dann wird bei jedem Lauf des Speicherbereinigers die Lokalität erhöht. Das Umkopieren erzeugt nur geringe Zusatzkosten.

Typanhäufen versucht die Vorteile von Anhäufen mit einem statistischen Verfahren auszunutzen, die Zusatzkosten jedoch zu vermeiden. Typanhäufen

wird auf Typen angewendet, deren pufferkritische Kantenfelder Verbunde des gleichen Typs referenzieren. Wir erweitern Typanhäufen und wenden es auf eine Menge Typen an, die sich gegenseitig über pufferkritische Kantenfelder referenzieren.

Typanhäufen platziert alle Instanzen eines Typs in einer gemeinsamen Speicherregion, einem *Haufen*. Dadurch liegen die Verbunde nebeneinander im Speicher und fallen in gemeinsame Pufferzeilen. Es ist jedoch nicht gewährleistet, dass Verbunde die sich gegenseitig referenzieren in eine gemeinsame Pufferzeile fallen.

Wird in nicht fragmentiertem Speicher alloziert, liefern viele Allokatoren bei aufeinander folgenden Speicheranfragen nach Speicher aus der gleichen Region Referenzen auf benachbarte Speicherstücke zurück. In dynamischen Datenstrukturen muss, nachdem ein Speicherstück alloziert wurde, die Referenz auf dieses Speicherstück in einem Verbund abgespeichert werden. Häufig wurde dieser Verbund kurz vorher alloziert. Wurde er direkt vorher alloziert, und werden die Verbunde in benachbarten Speicherstücken alloziert, können sie in eine gemeinsame Pufferzeile fallen.

Wendet man Typanhäufen auf Typen an, die ausschließlich pufferkritische Felder enthalten, fallen dadurch Verbunde mit pufferkritischen Feldern in gemeinsame Pufferzeilen. Bei normaler Allokation liegt eventuell neben der Instanz mit den pufferkritischen Feldern eine Instanz eines anderen Typs, dessen Felder unkritisch sind. Dann werden unkritische Felder mit der pufferkritischen Instanz in den Pufferspeicher geladen. Die Wahrscheinlichkeit, dass pufferkritische Felder geringe Zugriffsabstände haben, ist höher, als wenn unkritische und pufferkritische Felder in einer Pufferzeile liegen.

Das Anhäufen von Verbunden, die durch ein Kantenfeld verbunden sind, bringt zusätzliche Vorteile. Das Kantenfeld deutet darauf hin, dass Verbunde dieses Typs einen geringen Zugriffsabstand haben. Verbunde verschiedener Typen, die ähnlich pufferkritisch sind, haben rein statistisch betrachtet die gleichen Zugriffsabstände. Es ist jedoch nicht erkennbar, ob man von dem einen zum anderen navigieren kann, und diese verschiedenen Verbunde überhaupt mit einem geringen Zugriffsabstand zugegriffen werden können.

Typanhäufen erhöht also die Wiederverwendung und die Lokalität, indem es Felder mit möglicherweise geringen Zugriffsabständen in gemeinsamen Pufferzeilen platziert.

[GTZ98] suchen durch Analyse der Typstruktur eines Programms Behältertypen und platzieren einen gefundenen Behältertypen in einem eigenen Haufen. Unsere Optimierung optimiert solche Behältertypen auch,

allerdings nur wenn diese als pufferkritisch eingestuft werden. Wir wenden die Optimierung in weitaus mehr Situationen an. Wir machen weniger strukturelle Voraussetzungen an die Typen und platzieren mehrere Typen in einem Haufen, wenn die Analyse räumliche Wiederverwendung zwischen diesen vermutet.

6.1.1 Zuordnung von Typen zu Haufen

Dieses Kapitel beschreibt unseren Ansatz, in einem Programm Typen für Typanhäufen auszuwählen. Dieser berücksichtigt erstmals Informationen über Pufferprobleme der zu optimierenden Datenstrukturen. Wir bestimmen Mengen von Typen, die wir in einem Typhaufen platzieren wollen. Wir wählen ein zweistufiges Vorgehen, das wir unter Verwendung verschiedener Heuristiken iterieren, bis wir befriedigende Haufen gefunden haben.

Zuerst wählen wir mit einer Heuristik eine Teilmenge pufferkritischer Typen als Kandidaten für Haufen aus. Nun suchen wir in dieser Teilmenge nach Haufen. Finden wir keine, bestimmen wir mit einer schwächeren Heuristik eine größere Teilmenge pufferkritischer Typen und suchen darin nach Haufen. Dies wiederholen wir einige Male. Bevor die Heuristik zu allgemein wird, verzichten wir darauf, Typanhäufen auf das Programm anzuwenden.

Zunächst beschreiben wir, wie wir in einer gegebenen Menge pufferkritischer Typen nach geeigneten Haufen suchen.

Zuordnung einfacher Typen

Wir berechnen einen Graph, der die pufferkritischen Typen als Knoten, die pufferkritischen Kantenfelder als Kanten enthält. Typen, die das Ziel von Kanten sind, die von einem pufferkritischen Typ ausgehen, fügen wir dem Graph auch dann hinzu, wenn sie nicht pufferkritisch sind. Wir betrachten bei diesen Typen jedoch keine weiteren Kantenfelder. Wir nennen diesen Graph *Haufengraph* (Abb. 6.1). In diesem Graph wählen wir maximale starke Zusammenhangskomponenten als Typmengen für Typanhäufen aus. Eine starke Zusammenhangskomponente gewährleistet, dass von jedem Typ in der Komponente jeder andere über pufferkritische Kantenfelder erreicht werden kann.

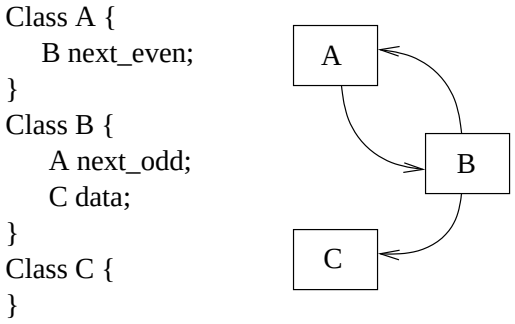

Abbildung 6.1: Ein Haufengraph. A und B können eine verkettete Liste bilden. Im Haufengraph sind sie eine starke Zusammenhangskomponente und ein potenzieller Haufen. Das Referenzfeld `data` auf C ist nicht rekursiv, C kommt also nicht in den Haufen.

Zuordnung in Vererbungshierarchien

Typen mit Vererbung können sich zyklisch referenzieren, ohne dass man das direkt an der Typrelation sieht. Gibt es ein Kantenfeld zwischen zwei Typen, so existieren auch Kantenfelder zwischen allen Untertypen des Ursprungstyps der Kante zu dem Zieltyp der Kante. Diese Felder wurden beim Auflösen impliziter Vererbung eingeführt.

Das Kantenfeld kann zur Laufzeit jedoch auch alle Untertypen des Zieltyps referenzieren. Gibt es im Haufengraph einen Pfad von diesem Untertypen zum Ursprungstyp, jedoch nicht vom eigentlichen Zieltyp der Kante, existiert ein Zyklus, der bisher im Haufengraph nicht repräsentiert ist. Daher fügen wir für jeden Zieltyp zusätzliche Kanten zu den Untertypen des Zieltyps in den Haufengraph ein. Ein Beispiel findet sich in Abb. 6.2.

Dies ähnelt der Problematik aus Kap. 5.2.1. Dort jedoch genügt es, die Untertyprelation in den Graphen aufzunehmen, da wir nur an rekursiven Kanten interessiert sind. Die Kanten, die die Untertyprelationen repräsentieren, werden bei der Auswertung der Zusammenhangskomponenten ignoriert. Hier interessieren wir uns für alle Knoten, die zu einem Zyklus beitragen. Verwenden wir die Untertyprelation, ist der Zieltyp eines Kantenfelds immer im Haufen vertreten. Verwenden wir statt der Untertyprelation die ‚durchgeschleiften' Kanten, die direkt auf die Untertypen verweisen, enthält der Haufen den Zieltyp nicht, der praktisch nicht in rekursiven Zugriffen vorkommen kann.

```
Class A {
}
Class B extends A {
   A f1;
}
Class C extends A {
   B f2; A f3;
}
Class D extends A {
   D f4; E f5;
}
Class E {
   D f6;
}
```

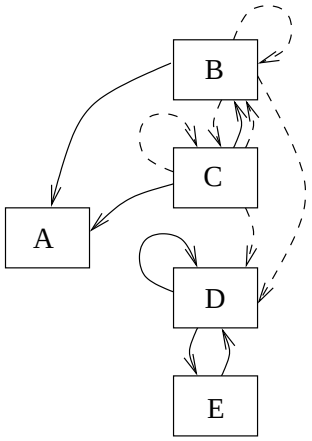

Abbildung 6.2: Ein Haufengraph bei Vererbung. Gestrichelte Kanten sind aufgrund der Vererbung eingeführt worden. Das Feld B.f1 erzeugt die Kante B → A. Es kann durch Vererbung auch eine Kante B → B, B → C oder B → D bedeuten. Der Graph zerfällt in zwei Haufen, B, C und D, E.

Würden wir die Zusammenhangskomponente im Graph aus Abb. 5.10(c) als Haufen verwenden, würde dieser A, B und C enthalten. Statt der Untertyprelation fügen wir hier eine Kante C → B ein, und erhalten als Haufen A und B.

Die Heuristiken

Wir verwenden eine dreistufige Heuristik, das heißt, wir suchen in höchstens drei Iterationen nach geeigneten Haufen. Die einzelnen Heuristiken unterscheiden sich in der Auswahl der Knoten und Kanten für den Haufengraph. Für die Heuristiken verwenden wir die Maße aus Kap. 5.3.4.

In der strengsten Heuristik verwenden wir nur Typen, deren Allokationshäufigkeit als pufferkritisch eingestuft wurde, und rekursive Kantenfelder, die entweder in ihrem Besitzertyp oder insgesamt als pufferkritisch eingestuft wurden. Die zweitstrengste Heuristik betrachtet alle Typen, beschränkt sich aber auf die oben genannten Kantenfelder. Die großzügigste Heuristik betrachtet zusätzlich alle rekursive Kantenfelder.

6.1.2 Haufenbildung zur Laufzeit

Die technische Haufenbildung hängt von der Laufzeitumgebung der übersetzten Programmiersprache ab. Grundsätzlich gehen wir davon aus, dass die Speicherverwaltung durch die Hochsprache festgelegt ist. Nur dann können wir Allokationen korrekt identifizieren und in der Zwischensprache geeignet darstellen.

Sieht die Hochsprache eine Halde vor, auf der Speicher unbeschränkt alloziert und freigegeben werden kann, schaffen wir für jeden Typhaufen eine eigene Halde. Für nicht angehäufte Typen verwenden wir die ursprüngliche Halde.

Enthält das Laufzeitsystem einen Speicherbereiniger nach dem Mark-and-Sweep- oder Referencecount-Algorithmus, sehen wir ebenfalls eine eigene Halde für jeden Typhaufen vor. Verwendet der Speicherbereiniger einen kopierenden Generationenalgorithmus, können wir eigene Generationen für die Typhaufen bilden. Ein Generationenalgorithmus alloziert Objekte in kleine Halden. Ist eine Halde voll, werden die nächsten Objekte in einer neuen, leeren Halde alloziert. Sind zu wenig leere Halden vorhanden, selektiert eine Heuristik eine Halde, in der Regel eine die schon lange voll ist, und kopiert die darin noch lebendigen Objekte in eine weitere Halde.

Mit Typanhäufen werden nicht alle neuen Objekte in der gleichen Halde alloziert, sondern der Speicherbereiniger sieht eine Halde für jeden Haufen vor.Die physikalischen Adressen der Speicherseiten eines Haufens sollten möglichst so gewählt werden, dass sie konfliktfrei auf den Puffer abgebildet werden.

6.1.3 Getrennte Übersetzung

Optimierungen der Datenanordnung setzen häufig voraus, dass alle Verwendungen einer Datenstruktur bei der Übersetzung bekannt sind. Dies ist bei getrennter Übersetzung selten.

Typanhäufen kann jedoch auch bei getrennter Übersetzung angewandt werden. Die Optimierung ändert lediglich die Allokationen im übersetzten Programmteil, die Schnittstelle zu Instanzen des optimierten Typs bleibt unverändert. Werden in einem externen Programmteil Instanzen dieses Typs alloziert, werden diese Instanzen nicht auf der separaten Halde alloziert, und haben daher keine erhöhte Lokalität. Die Korrektheit des Programms ist jedoch gewährleistet.

Kann das Laufzeitsystem dem Typ ansehen, wie er alloziert werden soll, können Instanzen des Typs auch bei Allokation in getrennt übersetzten Programmteilen auf dem Haufen platziert werden. Dies ist für Haufenbildung durch einen Speicherbereiniger leicht realisierbar.

6.2 Verbundteilen für Vererbungshierarchien

Verbundteilen ist eine Transformation, die die Effizienz von Anordnungsoptimierungen für Zeigeranwendungen wie Typanhäufen, Anhäufen oder Färben erhöht. Wir setzen Verbundteilen für Typanhäufen ein.

Typanhäufen erhöht die räumliche Lokalität von Verbunden. Enthalten die Verbunde unkritische Felder, reduziert dies die Lokalität.

Verbundteilen teilt einen Verbund in zwei (oder mehrere) Teile, die unabhängig voneinander im Speicher platziert werden können. Pufferkritische Felder werden von unkritischen Feldern getrennt. Der Teil der die pufferkritischen Felder enthält, qualifiziert sich eher für Typanhäufen als der ungeteilte Typ. Da die Typhaufen daraufhin weniger unkritische Felder enthalten, steigt die Wahrscheinlichkeit, dass Wiederverwendung und Lokalität entstehen.

Um einen Verbund zu teilen, werden zwei Verfahren vorgeschlagen. [CDL99] verwenden eine Indirektion, um auf den abgetrennten Teil zuzugreifen. [KF00, FK98] platzieren die Teile um n Adressen verschoben in einer Speicherregion. Der unkritische Teil kann mit einem großen aber konstanten Adressabstand (offset) zugegriffen werden. Damit dieses Verfahren funktioniert, müssen die Teile gleich groß sein. Ansonsten entsteht Verschnitt. Liegt dieser zwischen den unkritischen Teilen, kostet dies lediglich Speicherplatz. Liegt er zwischen den pufferkritischen Teilen, verschlechtert dies die Pufferleistung, da der ungenutzte Verschnitt mit in den Pufferspeicher geladen wird. Für dieses Verfahren ist ein angepasster Speicherbereiniger notwendig. Beide Verfahren verursachen zusätzliche Laufzeitkosten.

Wir beschreiben hier Lösungen, um Verbundteilen auf Vererbungshierarchien anzuwenden [Lin05]. Dabei ergeben sich zwei Probleme. Ein Verbund in einer Vererbungshierarchie muss auch nach dem Teilen kompatibel mit anderen Verbunden der Hierarchie sein. Neue Felder in einem Untertyp können die Performanz einer für einen Obertyp festgelegten Teilung beeinträchtigen. Das nächste Kapitel beschreibt mehrere Erweiterungen von Verbundteilen auf Vererbungshierarchien. Kap. 6.2.2 zeigt dann, wann wir welche der Erweiterungen anwenden.

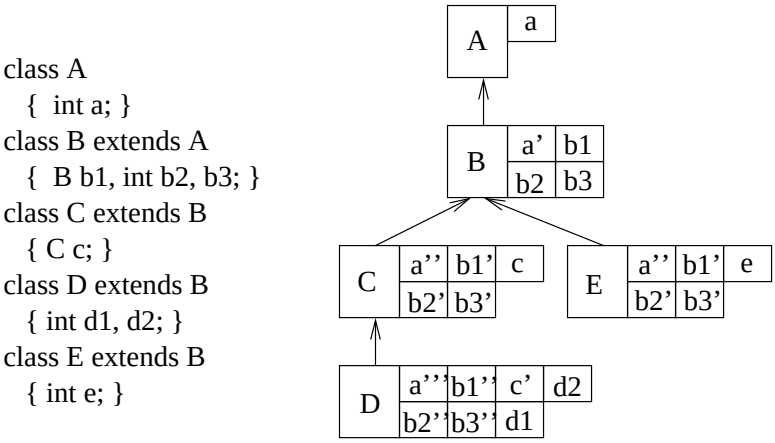

Abbildung 6.3: Eine Vererbungshierarchie. Felder $b1$ und c wurden als pufferkritisch eingestuft, die anderen sind alle unkritisch. Entgegen vorherigen Bildern wählen wir eine kompaktere Darstellung der Typhierarchie. Die Überschreibt-Relation stellen wir nur implizit dar: Felder mit gestrichenem Namen überschreiben Felder mit ungestrichenem Namen.

6.2.1 Teilen für Vererbungshierarchien

Wir nehmen im Folgenden an, das uns ein Typ T gegeben ist, der geteilt werden soll. Weiter ist die Teilmenge der pufferkritischen Felder gegeben. Zunächst betrachten wir Verbundteilen anhand dieses einen Typs, um den Algorithmus dann auf Vererbungshierarchien zu erweitern. Abb. 6.3 zeigt ein Beispiel für eine Vererbungshierarchie.

Verbundteilen ohne Vererbung

Wir betrachten zunächst einen Typ T ohne Ober- bzw. Untertypen.

Um T zu teilen, erzeugen wir einen neuen Typen T2 und verschieben alle unkritischen Felder von T nach T2. Wir fügen T ein neues Feld m_{ref} hinzu. Der Typ von m_{ref} ist ein Referenztyp auf T2, der den abgetrennten Teil von T beschreibt.

Methoden, die Instanzen des geteilten Verbunds verwenden, müssen der geänderten Anordnung angepasst werden. Dazu iterieren wir über die Rümpfe aller Methoden. Allokationsknoten werden um eine zweite Allokation für den abgetrennten Teil und die Initialisierung von m_{ref} ergänzt. Abb. 6.4 zeigt die Anpassung einer Allokation.

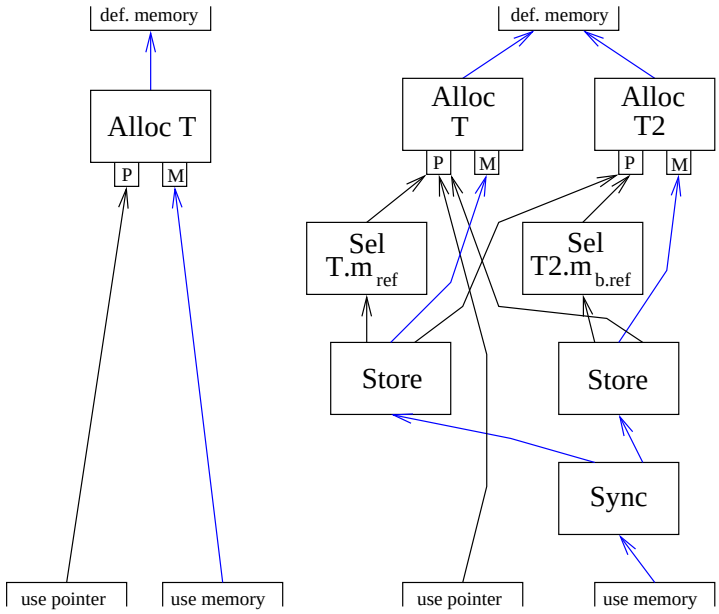

Abbildung 6.4: Anpassung einer Allokation. Die Abbildung zeigt auch die Initialisierung des weiter unten eingeführten Felds $m_{backref}$.

Feldzugriffe auf die verschobenen Felder werden um den Zugriff auf m_{ref} ergänzt. Abb. 6.5 zeigt die grundlegende Transformation eines Zugriffs. Ausnahmen durch den Zugriff fangen wir ab, indem wir den Zugriff nach einem vorhandenem expliziten Ausnahmetest platzieren, der für den Feldzugriff vorgesehen ist. Oder wir ordnen dem Laden der Referenz die Ausnahmeattribute des angepassten Feldzugriffes zu. Ist beides nicht möglich, müssen wir einen zusätzlichen expliziten Ausnahmetest einführen.

Da das Referenzfeld übersetzergeneriert ist, wissen wir, dass es für jede Instanz konstant ist und eine gültige Adresse enthält. Daher können wir für die erzeugten Operationen Elimination gemeinsamer Teilausdrücke anwenden. Zusätzlich wissen wir, dass diese Operation keine eigene Ausnahme erzeugen kann.

Verbundteilen ist nur möglich, wenn die Hochsprache das Layout eines Verbundtypen nicht festlegt. Dann hat der Übersetzer die Aufgabe, dieses Layout zu bestimmen. Ergeben sich dabei im pufferkritischen Teil Lücken, wird wieder ungebrauchter Speicher in den Pufferspeicher geladen. In diesem Fall kann man Felder, die eigentlich im unkritischen Teil platziert werden

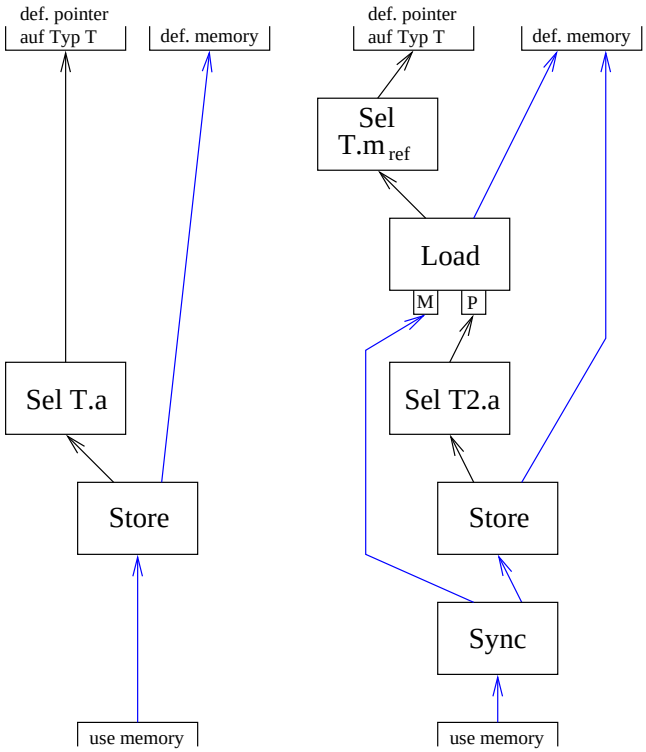

Abbildung 6.5: Anpassung eines schreibenden Feldzugriffes. Analog für lesende Zugriffe.

sollen, im pufferkritischen Teil unterbringen, ohne den Speicherbedarf des pufferkritischen Teils zu erhöhen. Dies reduziert Indirektionen, Fehlzugriffe und Speicherbedarf.

Typsicheres Verbundteilen

Typsicheres Verbundteilen erweitert das Teilen eines Verbunds auf Vererbungshierarchien. Diese Erweiterung garantiert, dass der geteilte Verbund sich ohne weiteres in die Vererbungshierarchie einfügt. Annahmen des Programms über die Typumwandelbarkeit zwischen einzelnen Verbunden der Hierarchie werden nicht verletzt. Wir stellen zwei Varianten typsicheren Verbundteilens vor.

Atomares, typsicheres Verbundteilen ist die triviale Erweiterung von Verbundteilen auf Vererbungshierarchien. Hat der geteilte Typ Obertypen, so

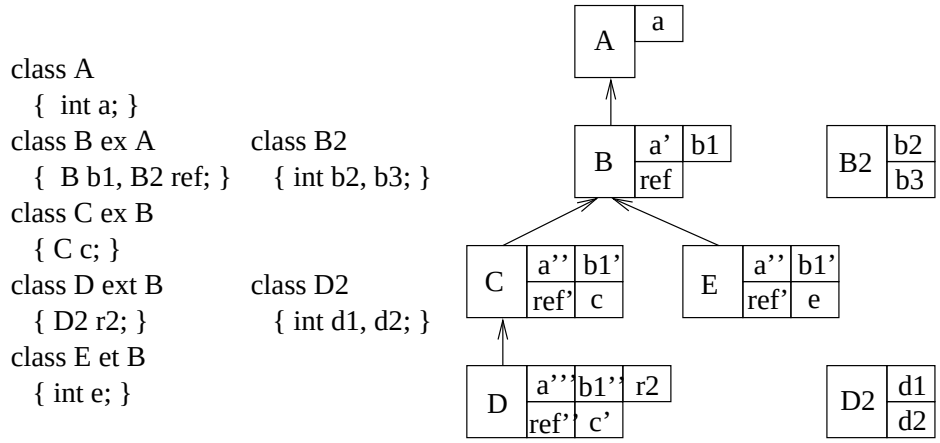

Abbildung 6.6: Atomares, typsicheres Verbundteilen für Beispiel 6.3. Nur die Felder $b2$ und $b3$ wurden abgetrennt. Bei einer zweiten Anwendung wurden $d1$ und $d2$ abgetrennt. Es gibt in D zwei Referenzfelder ref und $r2$.

werden keine Felder, die von einem Obertyp geerbt sind, in den abgetrennten Teil verschoben. Untertypen des geteilten Typs werden genau wie der geteilte Typ geteilt. Jeder bekommt einen eigenen abgetrennten Teil und erbt das Feld m_{ref}. Im abgetrennten Teil werden nur Felder platziert, die Felder des abgetrennten Obertypen überschreiben. Die abgetrennten Untertypen bekommen eine Untertyprelation analog zu ihrem ursprünglichen Typ. Sind alle verschobenen Felder in den Untertypen von der Ausprägung ‚geerbt', müssen die abgetrennten Typen nicht dargestellt werden. Dieses Verfahren wird rekursiv für alle Untertypen angewandt. Allokationen und Feldzugriffe werden für alle Untertypen angepasst. Abb. 6.6 zeigt atomares, typsicheres Verbundteilen.

Atomares, typsicheres Verbundteilen berücksichtigt nicht, dass ein Untertyp neue unkritische Felder einführen könnte. Dadurch kann solch ein Untertyp, der durchaus pufferkritische Felder von seinem Obertyp erbt, sich nicht für weitere Optimierungen qualifizieren. Er wird bei Typanhäufen nicht dem Haufen mit dem Obertyp zugeschlagen. Dies leistet adaptives, typsicheres Verbundteilen. Atomares, typsicheres Verbundteilen entspricht am ehesten den in der Literatur beschriebenen Algorithmen, die Verbunde auf Quellsprachebene optimieren.

Führt ein Untertyp eines geteilten Typen weitere unkritische Felder ein, kann man atomares, typsicheres Verbundteilen erneut anwenden. Man erhält

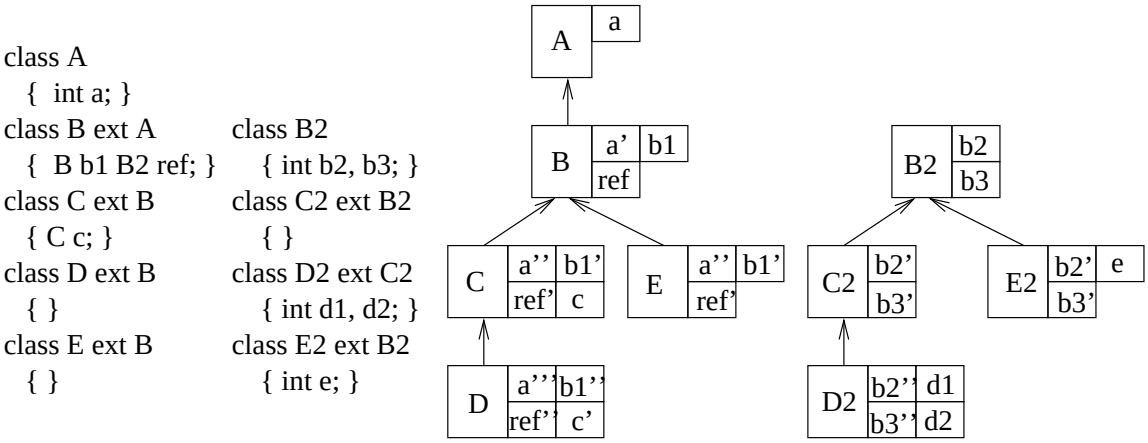

Abbildung 6.7: Adaptives Verbundteilen für Beispiel 6.3. Felder $d1$, $d2$ und e werden ebenfalls verschoben.

dadurch allerdings mehrere Referenzfelder im pufferkritischen Teil.

Adaptives, typsicheres Verbundteilen passt die Menge pufferkritischer Felder den Verwendungshäufigkeiten der neuen Felder in den Untertypen an. Zunächst wird diese Menge wie in Kap. 6.2.1 initialisiert. Dann werden alle Felder, die in einem Untertypen initial definiert sind, betrachtet. Diese überschreiben kein Feld aus einem Obertyp. Sind diese pufferkritisch, werden sie ebenfalls zur Menge pufferkritischer Felder hinzugefügt. Durch diese Aufteilung in pufferkritische und unkritische Felder wird eine gemeinsame Teilung aller Untertypen von T erreicht. Abb. 6.7 zeigt ein Beispiel.

Durch adaptives, typsicheres Verbundteilen wächst der pufferkritische Teil des Verbunds nur noch geringfügig und nur um pufferkritische Felder. Der Untertyp kann sinnvoll auf einem Typhaufen platziert werden. Neue pufferkritische Felder im Untertypen werden bevorzugt behandelt. Das eingeerbte Referenzfeld wird zum Teilen des Untertypen wiederverwendet, wodurch man sich gegenüber atomarem, typsicheren Verbundteilen Felder im pufferkritischen Teil spart.

Optimales Verbundteilen

Bei typsicherem Verbundteilen werden alle nicht geteilten Obertypen in den pufferkritischen Teil eingeerbt, um Typkompatibilität zu gewährleisten. Dies betrifft sowohl die Obertypen von T, sowie weitere Obertypen der Untertypen von T. Da die Obertypen nicht geteilt werden, enthalten sie keine

pufferkritischen Felder. Diese unkritischen Felder stören die Pufferperformanz des geteilten Typs.

Optimales Verbundteilen vermeidet, dass der pufferkritische Teil durch eingeerbte, unkritische Felder verunreinigt wird. Es ist also insofern optimal, das kein von der Heuristik als unkritisch eingestuftes Feld im cachekritischen Teil verbleibt. Dazu vererben Obertypen von T an den abgespaltenen Teil des Typs, an T2. Dies bewirkt, dass T und seine Obertypen nicht mehr typkompatibel sind. Wir müssen explizite Typanpassung einführen. Wird ein T zu einem Obertyp von T umgewandelt, muss die Referenz auf T durch die Referenz auf den abgespaltenen Typ ersetzt werden. Die Referenz auf den abgespaltenen Teil können wir dem bestehenden Feld m_{ref} entnehmen. Bei einer umgekehrten Typanpassung müssen wir die Referenz auf den abgespaltenen Teil durch eine Referenz auf T ersetzen. Dazu müssen wir ein neues Feld $m_{backref}$ im abgespaltenen Teil einführen, das die Referenz auf T enthält. Durch diese Typanpassungen steigt der Laufzeitaufwand und der Speicherbedarf. Zusätzlich kann es nötig werden, Felder zu replizieren. Der unkritische Teil benötigt eine eigene Referenz auf die Sprungtabelle damit er mit seinem Obertyp kompatibel ist. Diese Referenz ist jedoch häufig pufferkritisch, so dass sie ebenfalls im pufferkritischen Teil platziert werden sollte. Da der Inhalt dieses Feldes konstant ist, kann es in beiden Teilen vorgesehen werden, ohne während der Laufzeit Synchronisationsaufwand zu generieren.

Optimales Verbundteilen erhöht also Speicherbedarf und bedeutet zusätzliche Zugriffe, dafür enthält der pufferkritische Teil ausschließlich pufferkritische Felder, die zu erwartende Pufferleistung ist maximal. Abb. 6.8 zeigt ein Beispiel.

Für optimales Verbundteilen müssen wir explizite Typanpassungen einfügen. Daher ist optimales Verbundteilen nur dann möglich, wenn alle Typanpassungen zum Optimierungszeitpunkt bekannt sind. In unserer Zwischensprache Firm werden Typanpassungen durch die Cast-Operation dargestellt, siehe Tabelle 4.1. Die explizite Typanpassung kann nur auf Werte angewandt werden, die tatsächlich eine Instanz des optimierten Typs darstellen. Programmiersprachen definieren in der Regel eine ausgezeichnete Variable. Die konstante Referenz auf diese Variable, NULL in Java oder C, ist zu allen Referenztypen kompatibel, darf aber nicht dereferenziert werden. Auf diese Referenz dürfen wir die explizite Typanpassung nicht anwenden. Das Laden der Referenzfelder m_{ref} bzw. $\mathrm{m}_{backref}$ würde eine Ausnahme auslösen, die im ursprünglichen Programm hier nicht auftreten würde. Wir müssen solche Werte durch Tests ausschließen, wenn wir dem Programmcode nicht ansehen

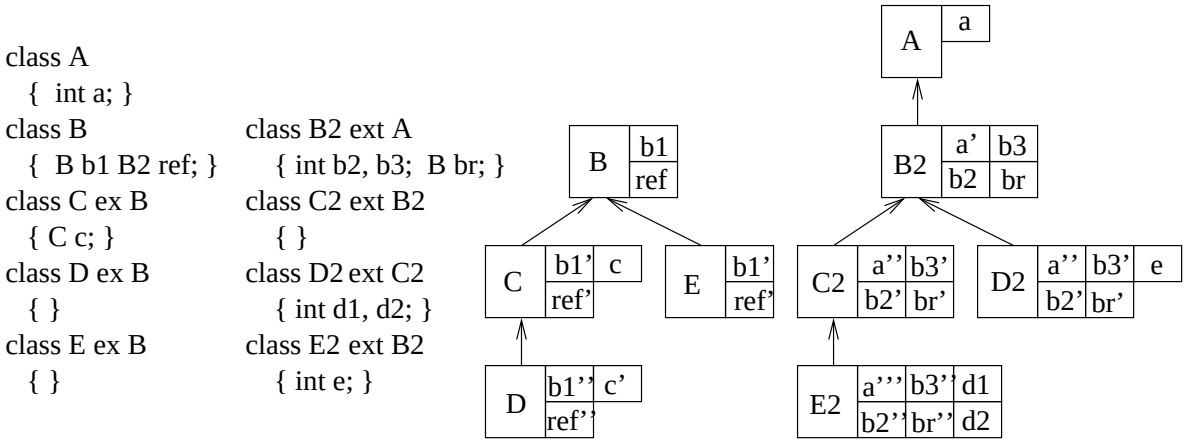

Abbildung 6.8: Optimales Verbundteilen für Beispiel 6.3. A vererbt a an den abgespaltenen Teil. Der pufferkritische Teil enthält neben ref nur noch pufferkritische Felder. Das zusätzliche Feld *br* verbraucht Speicher, ist aber im unkritischen Teil und beeinflusst die Pufferperformanz damit nur geringfügig.

können, das das Auftreten dieser Werte unmöglich ist. Abb. 6.9 zeigt den Einbau einer expliziten Typanpassung bei einem `Cast`.

Durch polymorphe Methodenaufrufe entstehen implizite, dynamische Typanpassungen. Ein polymorpher Aufruf gibt typischer Weise als ein Argument die Variable an, anhand deren dynamischen Typs die aufgerufene Methode ausgewählt wurde, das `self` Argument. An der Aufrufstelle ist der statische Typ für dieses Argument angegeben. Zur Laufzeit können hier jedoch Variablen jedes Untertyps dieses statischen Typs übergeben werden. Die aufgerufene Methode jedoch, die dynamisch ausgewählt wurde, ist für einen festen statischen Typ ausgelegt. Da nach optimalem Verbundteilen die Verbunde nicht mehr automatisch typkompatibel sind, muss hier eine dynamische Typanpassung eingeführt werden.

Dies erreichen wir, indem wir Umschlagsmethoden generieren. Diese überschreiben die Methoden ungeteilter Obertypen. Sie wenden die dynamische Typanpassung an und rufen dann statisch die gewünschte Methode auf. Da der `self` Parameter immer einen gültigen Wert hat, muss kein NULL Test ausgeführt werden. Es entstehen zusätzliche Einträge in der Sprungtabelle für die ursprünglichen Methoden.

Wir kennen, wie oben gefordert, alle Typanpassungen für Typen die durch optimales Verbundteilen optimiert werden. Wird der Typ nie zu einem seiner

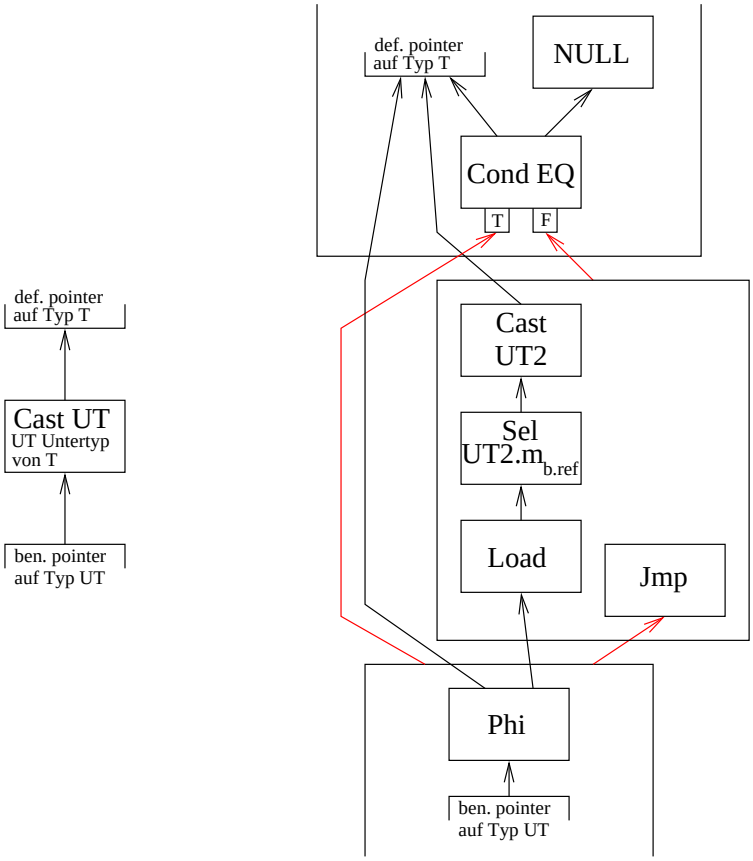

Abbildung 6.9: Anpassung einer Typanpassung ungeteilter Obertyp T → geteilter Untertyp UT. Analog für Anpassung Untertyp → Obertyp.

Obertypen umgewandelt, muss er nicht mit den Obertypen kompatibel sein. Stellen wir dies fest, erzeugen wir weder das Feld $m_{backref}$ noch Umschlagsmethoden. Kann ein Untertyp zu einem weiteren Obertyp umgewandelt werden, reicht es, hier dieses Feld und Umschlagsmethoden einzuführen. Wir nennen diese Variante *inkompatibles, optimales Verbundteilen*. Die Umschlagsmethode kann Subjekt weiterer Optimierungen sein. So kann die eigentlich aufzurufende Methode offen eingebaut werden. Dadurch wird der zusätzliche Methodenaufruf gespart und man kann die Typinformation durch den Cast Knoten zur Optimierung verwenden.

```
class A {
  void m(A self);
}
class B extends A {
  void m(B self);
}
```

(a)

```
class B {
  void m(B self);
  B2 ref;
}
```

```
class A {
  void m(A self);
}
class B2 extends A {
  void m_wrap(A self) {
     B.m(((B2) self).backref);
  }
  B backref;
}
```

(b)

Abbildung 6.10: Einfügen einer Umschlagsmethode. B wird geteilt, A nicht. (a) vor dem Teilen, (b) nach dem Teilen. Dieser Pseudocode stellt die `self` Parameter explizit dar. Wird auf einer Variablen des Typs A, die eine Referenz auf ein B enthält, die Methode m() aufgerufen, so wird dieser Methode nach dem Teilen eine Referenz auf B2 übergeben. Durch Einführen der Methode m_wrap(), die m() aus A überschreibt, wird das Argument self angepasst. Der Aufruf an B.m() kann statisch erfolgen.

6.2.2 Auswahl zu teilender Verbunde

Verbundteilen verursacht deutlich mehr zusätzliche Laufzeitkosten als Typanhäufen. Daher müssen zu teilende Typen mit strengeren Grenzwerten ausgewählt werden.

Auswahl einzelner Typen

Kandidaten für Verbundteilen sind Typen, die Felder enthalten, die als Kanten im Haufengraph in Frage kommen, die aber selbst nicht ausreichend pufferkritisch sind. Da die Bewertung des Typen aus der Bewertung der Felder berechnet wird, hat dieser Typ weitere, unkritische Felder.

Wir sortieren die Felder nach ihrer Zugriffshäufigkeit (#Zugriffe/#Allokationen). Wir berücksichtigen alle Felder, können aber die Zugriffshäufigkeit von Kantenfeldern etwas höher bewerten. Wir teilen den Typ, wenn die Menge Felder, die einen hohen Anteil (95% - 99.9%) aller Zugriffe bedeutet, nur wenige Felder umfasst. Verwenden wir eine Indirektion zum abgetrennten Teil, sollte der abgetrennte Teil größer sein als der Speicherbedarf für die Indirektion. Um so größer der abgetrennte Teil, und um so kleiner der pufferkritische Teil ist, um so eher lohnt sich Verbundteilen.

Wie folgende Überlegung zeigt, darf der abgetrennte Teil nur sehr wenige Zugriffe aufzeigen: Wenn jeder 20. Zugriff auf ein Feld ein Fehler ist, und wir durch die Optimierung 20% aller Fehler vermeiden, sparen wir einen Fehler auf 100 Zugriffe. Gehen wir davon aus, dass ein Fehler ungefähr soviel kostet wie der zusätzliche Zugriff durch die Indirektion, darf jeder 100. Zugriff eine Indirektion benötigen. Also dürfen 1% der Zugriffe auf den abgetrennten Teil zugreifen.

Auswahl in einer Vererbungshierarchie

In Vererbungshierarchien ist die Auswahl zu teilender Typen schwieriger, da mit einem Typen auch seine Untertypen geteilt werden. Wir suchen eine Aufteilung der Felder, die für alle Typen der zu teilenden Teilhierarchie geeignet ist.

Damit ein Feld unabhängig vom tatsächlichen Typ zugegriffen werden kann, müssen möglichst viele Typen in der Hierarchie, die das Feld erben, den gleichen Zugriffsmechanismus für das Feld haben. Ist das Feld in einem geteilten Typ im direkt referenzierten Teil des Verbunds, in einem anderen Typ in dem durch die Indirektion erreichbaren, muss man abhängig vom dynamischen Typ die Indirektion auflösen. Wird jeder Typ anders optimiert, reicht eine explizite Typanpassung wie bei optimalem Verbundteilen nicht, jeder Zugriff bedingt die Kosten eines expliziten Typtest. Zusätzlich können wir ein Feld nur in seinem Verhalten bezüglich all seiner Vorkommen bewerten. Daher spalten wir ein Feld nur in dem Typ ab, in dem es eingeführt wird.

Der Typ, in dem eine pufferkritisches Feld eingeführt wird, muss nicht unbedingt von Verbundteilen profitieren, wie Abb. 6.11 zeigt. Er kann entweder nur wenig unkritische Felder enthalten, so dass Verbundteilen nicht nötig ist. Oder es existieren von diesem Typen nur wenige Instanzen, so dass durch das Teilen kein entscheidender Laufzeitgewinn zu erwarten ist. Ergänzt ein Untertyp den Typ um viele unkritische Felder, und werden von diesem Untertyp, oder einem seiner Untertypen, viele Instanzen erzeugt, kann es dadurch rentabel werden, den ursprünglichen Typ zu teilen.

Ein Sonderfall ist, wenn ein Typ ausschließlich pufferkritische Felder enthält. Dann wenden wir typsicheres Verbundteilen für problematische Untertypen an. Wir vermeiden dadurch, dem Typ ein Referenzfeld zu geben.

Dies zeigt, dass wir bei der Entscheidung über Verbundteilen den

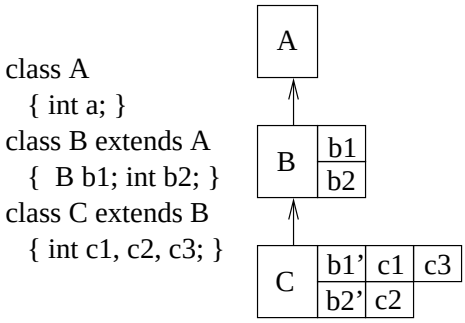

Abbildung 6.11: Feld $b1$ ist pufferkritisch, alle anderen nicht. Typ B zu teilen lohnt sich nicht, da der pufferkritische Teil nicht kleiner wird. Gibt es von Typ C wenig Instanzen, lohnt sich auch hier Verbundteilen nicht. Gibt es viele Instanzen von Typ C, lohnt es sich B zu teilen.

Speicherbedarf der unkritischen Felder zur Laufzeit abschätzen müssen. Benötigen diese viel Speicher, da viele Instanzen mit den Feldern alloziert werden, lohnt sich Verbundteilen. Wir können den Speicherbedarf mit dem Produkt aus Feldgröße und der Anzahl akkumulierter Allokationen des einführenden Typs abschätzen.

Um zu teilende Typen zu bestimmen, gehen wir wie folgt vor. Wir berechnen für jeden Typ zwei Mengen: die Menge pufferkritischer Felder, und die Menge unkritischer Felder. Für jede Menge berechnen wir zusätzlich den potentiellen Speicherbedarf: Dieser ergibt sich aus der Summe des Speicherbedarfs für die Felder.

Wir berechnen diese Mengen rekursiv auf dem Typgraphen. Wir initialisieren die Mengen für einen Typ, indem wir die Mengen seiner Obertypen vereinigen. Nun fügen wir alle im Typ neu definierten Felder den Mengen hinzu.

Da wir ausschließlich Typen teilen wollen, die sich nach dem Teilen für Typanhäufen qualifizieren, bewerten wir die Menge pufferkritischer Felder mit der Heuristik, die entscheidet, ob sich ein Typ für den Haufengraph qualifiziert. Zusätzlich schätzen wir ab, ob die unkritische Feldermenge einen ausreichenden Speicherbedarf hat. Trifft beides zu, markieren wir den Typ als Kandidat für Verbundteilen.

Nun betrachten wir die eingeerbten Felder in der pufferkritischen Kantenmenge und markieren die Typen, in denen diese definiert sind, ebenfalls für Verbundteilen.

Entscheidung zwischen adaptivem und optimalem Verbundteilen Optimales Verbundteilen steigert potentiell die Puffereffizienz, die expliziten Typanpassungen kosten aber garantiert zusätzliche Laufzeit gegenüber typsicherem Verbundteilen. Um sich für einen der beiden Algorithmen zu entscheiden, muss man die Zusatzkosten durch die Typanpassungen berücksichtigen.

Obertypen dienen oftmals um eine gemeinsame Schnittstelle für mehrere Implementierungen zu definieren. In diesem Fall können die Obertypen keine eigenen Felder enthalten. Dann lohnt sich optimales Verbundteilen nicht, da keine Felder vom nicht geteilten Obertypen geerbt werden.

Definiert der Obertyp eigene Felder, müssen wir die Kosten der Typanpassungen abschätzen. Dazu sammeln wir mit der in Kap. 5.3.1 beschriebenen Analyse alle Cast-Operationen zu einem Typ auf. Die Cast-Operation gibt den Zieltyp der Anpassung an. Den Quelltyp finden wir, indem wir in der Zwischendarstellung die Eingangskante der Cast-Operation bis zu ihrer Quelle zurück verfolgen. Diese Quelle spezifiziert in Firm den Typ des erzeugten Wertes. Anhand der Ausführungshäufigkeit der Cast-Operationen können wir dann die Häufigkeit von Typanpassungen abschätzen.

Gibt es keine Typanpassung von einem Obertypen in den Untertypen, können wir sogar das Referenzfeld, das nur für diesen Cast benötigt wird, vermeiden.

Der Aufwand einer expliziten Typanpassung entspricht in etwa dem eines Zugriffs durch eine Indirektion. Daher schlagen wir die Typanpassungen als Feldzugriffe der akkumulierten Anzahl Zugriffe auf die Menge abzuspaltender Felder zu. Da die relevanten Typanpassungen von der Menge zu teilender Typen abhängt, können wir dies nicht mit der Mengenfindung kombinieren. Daher entscheiden wir über adaptives oder optimales Verbundteilen, nachdem wir zu teilende Typen identifiziert haben. Wenn sich diese unter Berücksichtigung der Typanpassungen immer noch für Verbundteilen qualifizieren, wenden wir optimales Verbundteilen an.

6.2.3 Verbundteilen als Übersetzeroptimierung oder Quelltransformation?

Verbundteilen als Übersetzeroptimierung zu implementieren bringt verschiedene Vorteile gegenüber einer Quelltransformation.

Wenn eine Quellsprache implizite Obertypen spezifiziert, wie in Java zum

Beispiel die Klasse `Object`, ist jede Teilung nur eine Mischform aus typsicherem und optimalem Verbundteilen. Beide neuen Typen erben zumindest von dem impliziten Obertyp. Dadurch werden zusätzliche Felder, die die Sprache implizit vorsieht, mehrfach erzeugt.

Bei einer Quelltransformation fällt es schwer, automatisch die nötigen Typanpassungen für optimales Verbundteilen einzuführen. Da die Referenzfelder als reguläre Felder der Hochsprache definiert werden müssen, bedingen sie den gleichen Zusatzaufwand wie der Zugriff auf normale Felder. In Java müssen zum Beispiel vor einem Feldzugriff von der Sprache vorgegebene Ausnahmen getestet werden. Diese entfallen bei einer Übersetzeroptimierung, da per Konstruktion klar ist, dass das Referenzfeld nie eine ungültige Referenz enthält.

Weiter können wir bei einer Übersetzeroptimierung durch den Übersetzer generierte Funktionalität vermeiden. So bekommen zwei Hochsprachtypen jeweils eine eigene Sprungtabelle und einen eigenen Konstruktor. Bei einer Übersetzertransformation benötigen wir nur eine Sprungtabelle und einen Konstruktor. Bei typsicherem Verbundteilen benötigt der abgetrennte Teil nie eine Referenz auf die Sprungtabelle. Bei optimalem Verbundteilen kann dies nach Bedarf entschieden werden. Beide Teile des Typs brauchen zusammen nur eine Sprungtabelle. In Umschlagsmethoden muss kein NULL Test erzeugt werden, und der Aufruf in der Umschlagsmethode kann statisch erfolgen.

Der Übersetzer kann Information über die automatisch eingefügten Typen an einen Speicherbereiniger weitergeben. Beide Teile eines geteilten Typs leben gleich lang. Daher muss der Speicherbereiniger nur die Lebendigkeit des ursprünglichen Teils ermitteln. Zum Beispiel wird nur ein Referenzzähler benötigt.

Funktionalität, die durch den Übersetzer eingebaut wird, kann einfacher optimiert werden, da zum Beispiel keine Aliase mit dem Rest des Programms bestehen können. Daher können wir, wie in Kap. 6.2.1 beschrieben, die Zugriffsoperationen über das Referenzfeld leicht optimieren. Dazu können wir die bereits im Übersetzer implementierten Analysen und Optimierungen verwenden. Ähnliche Analysen und Optimierungen auf Quellebene zu implementieren ist aufwendig und kompliziert. In der Regel sehen Hochsprachen nicht die notwendigen Annotationen vor, um das Wissen über die eingebaute Funktionalität an den Übersetzer zu übermitteln.

6.2.4 Getrennte Übersetzung

Verbundteilen ist nicht transparent für externe Programmteile. Ein geteilter Typ kann eine Schnittstelle zu Teilen des Programms, die bei der Übersetzung des Programms nicht bekannt sind, haben. In diesen Teilen kann weder die Allokation, noch der Zugriff auf Felder angepasst werden.

Dies kann durch eine rein funktionale Schnittstelle gelöst werden, in der auf jedes Feld durch eine Methode zugegriffen wird. Diese Methoden werden dann mit dem übersetzten Typ angepasst. Ist dies nicht möglich, da die Definition der Schnittstelle nicht beeinflusst werden kann, muss ein Analyse sicherstellen dass der Typ nicht extern sichtbar ist.

Kapitel 7

Experimente

Im Folgenden untersuchen wir mit Experimenten die Qualität der Analyse und die Verbesserungen der Pufferleistung durch die Optimierung.

Zunächst stellen wir die Versuchsumgebung und die Testprogramme vor. Wir untersuchen vier verschiedene Kombinationen unserer Optimierungen. Wir messen die Laufzeit und simulieren das Pufferverhalten der optimierten Programme. Um die Qualität der Analyse zu bewerten führen wir unsere Optimierungen zum Vergleich auch auf Basis von Profildaten durch.

7.1 Versuchsaufbau

7.1.1 Der verwendete Übersetzer

Wir verwenden den Übersetzer jack[BBGL05], der Java in ausführbaren Code übersetzt. Jack basiert auf dem Frontend des jikes [jik] Übersetzers von IBM. Aus dem abstrakten Syntaxbaum des Frontends wird die Zwischensprache Firm [TLB99] aufgebaut. Auf der Zwischensprache werden Standardoptimierungen wie Konstantenfaltung, Elimination Toten Codes, einfaches Auflösen von Polymorphie, schnelle Typanalyse [BS96] oder Offener Einbau ausgeführt. Dann führen wir die Pufferoptimierung und die dafür nötigen Analysen durch. Danach werden hochsprachnahe Konstrukte aus der Zwischensprache entfernt und nochmals Optimierungen durchgeführt. Ein mit dem Codegeneratorgenerator cggg [Boe05] erzeugter Codegenerator erzeugt aus der Zwischensprache assemblerähnlichen C-Code, der vom systemeigenen C-Übersetzer in ausführbaren Code übersetzt wird.

Jack repräsentiert Java Klassen als Verbundtypen mit Vererbung. Da jack Java in ausführbaren Code übersetzt, baut er verschiedene Funktionalitäten, die in einer standardkonformen Java Umgebung von der virtuellen Maschine geleistet werden, direkt in den Zielcode ein. Dies geschieht beim Aufbau der Zwischensprache. Dieser Code wird teilweise direkt in übersetzte Methoden eingebaut und teilweise in Form zusätzlicher, direkt in der Zwischensprache formulierter Methoden bereitgestellt. Dadurch kann dieser Code mit den im Übersetzer vorgesehenen Algorithmen speziell für das übersetzte Programm optimiert werden. Ein geringer Teil der Funktionalität der virtuellen Maschine wird durch zusätzlichen, nativen Code bereitgestellt, der an die Ausgabe des Übersetzers gebunden wird.

Zum Beispiel baut jack für jeden Reihungsfeldtyp einen eigenen Verbundtypen mit Vererbung auf, der Untertyp von einem generischen Verbundtyp für Reihungen ist. Dieser generische Reihungsverbundtyp enthält lediglich das von Java benötigte Reihungslängenfeld. Die Untertypen enthalten zusätzlich ein Feld, das eine ganze Reihung des Feldtyps enthält. Insbesondere wird für jede Java Klasse, die Feldtyp einer Reihung ist, ein eigener Reihungsverbundtyp konstruiert. Dadurch kann auf diese Reihungsverbundtypen sehr spezifisch Typanhäufen angewendet werden. Diese Typen qualifizieren sich nicht für Verbundteilen, da sie nur zwei Felder enthalten.

Wir führen jack im Modus für Ganzprogrammübersetzung aus. Dann übersetzt jack das Quellprogramm mit allen Klassen der Anwendung und den benötigten Klassen aus der Java Anwenderprogrammierschnittstelle (API) in einem Lauf. In diesem Modus unterstützt der Übersetzer einige Java Eigenschaften nicht mehr vollständig, wie zum Beispiel Reflexion und dynamisches Nachladen von Klassen. Dadurch hat der Übersetzer mehr Freiraum Datenstrukturen zu optimieren. Er nimmt an, dass er alle Zugriffe auf Felder und alle Allokationen kennt. Datenstrukturen, die im nativen Code verwendet werden, sind dem Frontend bekannt und werden von diesem als extern sichtbar und damit nur eingeschränkt optimierbar gekennzeichnet.

7.1.2 Die verwendeten Testprogramme

Als Testprogramme verwenden wir die Olden Testprogramme [HRCR95].

Die Olden Testprogramme wurden bereits in mehreren Arbeiten, die die Pufferleistung von Zeigeranwendungen optimieren, verwendet [LM96, LM99b, CHL99, SGF+02]. Sie liegen ursprünglich in C vor, aber [Cah02] übersetzte sie für seine Arbeit nach Java und nennt sie jOlden. Tabelle 7.1

Tabelle 7.1: Die Olden Testprogramme. (Fehlzugriffe in Prozent aller Zugriffe.)

Name	verwendete Datenstruktur	Zeilen	Fehlzugriffe
BH	8-fach Baum, verkettete Liste	1141	1.76
Bisort	Binärbaum	340	3.49
Em3d	verkettete Liste	464	13.49
Health	4-fach Baum, verkettete Liste	576	12.46
MST	Hashtabelle	473	24.37
Perimeter	4-fach Baum	751	1.66
Power	Baum	765	0.19
TreeAdd	Binärbaum	198	5.58
TSP	Binärbaum, verkettete Liste	544	3.10
Voronoi	Binärbaum	1000	0.62

zeigt die Datenstrukturen, die die Programme verwenden. Die Tabelle zeigt weiter die Anzahl der Programmzeilen, sowie die Fehlzugriffsrate eines unoptimierten Programmes in Prozent. Zum Beispiel sind bei BH 1,76% aller Zugriffe Fehlzugriffe. Dies zeigt dass insbesondere Em3d, Health, MST und TSP Verbesserungen durch eine Pufferoptimierung zeigen sollten.

Mit den Olden Testprogrammen zeigen wir das Potential dieser Arbeit für verschiedene Datenstrukturen. Die Programme sind verhältnismäßig klein. Da jedoch mit jedem Testprogramm die komplette Java API mit übersetzt wird, enthält jede Übersetzungseinheit eine Vielzahl von Klassen mit tiefen Vererbungshierarchien. Unter diesen Klassen sind solche, aus denen große Datenstrukturen aufgebaut werden können. Da diese von den Testprogrammen aber nicht genutzt werden, stellt die Übersetzungseinheit einer Heuristik eine durchaus ernsthafte Aufgabe. Die pufferkritischen Datenstrukturen müssen unter vielen unkritischen Datenstrukturen ausgewählt werden.

Wir haben unseren Übersetzer auch mit größeren Programmen wie den Javagrande Testprogrammen [BSW+00] getestet. Diese bieten unserer Optimierung jedoch keine geeigneten Probleme. Daher betrachten wir diese nicht weiter.

7.1.3 Messmethodik

Um die in dieser Arbeit vorgeschlagenen Optimierung zu bewerten, messen wir verschiedene Größen.

Die Pufferperformanz eines Programms zeigt sich am deutlichsten in der Anzahl der Pufferfehler bzw. im Verhältnis der Fehler zu der Anzahl aller Speicherzugriffe. Die Anzahl Speicherzugriffe eines Programms und die Größe des im Programmverlauf benötigten Speichers charakterisiert auch die Abhängigkeit eines Programms von der Speicherhierarchie des ausführenden Systems. Durch unsere Optimierungen werden zusätzliche Speicherzugriffe und Speicherbedarf eingeführt. Dies charakterisiert die Zusatzkosten durch die Optimierung. Zur Messung der Pufferperformanz verwenden wir eine Simulation.

Eine Puffersimulation

Den Speicherbedarf, die Zugriffe und die Fehler messen wir mit einer Puffersimulation. Dazu haben wir die Codegeneratorspezifikation so geändert, dass diese zusätzlich zu jeder Speicherallokation und jedem Speicherzugriff einen Aufruf an die Simulation generiert. Die Simulation selbst basiert auf der Implementierung von *cachegrind* [SN03], einem Werkzeug zum Messen der Pufferperformanz eines übersetzten Programms. Durch den direkten Einbau der Simulation in das Zielprogramm können wir jedoch genauere Aussagen erlangen, da wir zum Beispiel aus der Übersetzung wissen, auf welches Feld ein Zugriff zugreift. Die Meßdaten lassen sich so mit der Zwischendarstellung in einem zweiten Übersetzerlauf assoziieren. Zusätzlich ist die Simulation durch den direkten Einbau deutlich schneller. Wir simulieren einen 64KB, 2-fach assoziativen Pufferspeicher mit 64B großen Zeilen.

Laufzeitmessungen

Laufzeitmessungen führen wir im Single User Modus auf einem AMD Athlon 1400MHz, Stepping 4 mit 256KB Pufferspeicher durch. Die Experimente werden unter SUSE Linux 2.6.11.4 mit gcc 3.3.5 ausgeführt. Wir messen die Laufzeit mit der Methodes `System.currentTimeMillis()` aus der Java API, die von jack auf die Systemroutine `gettimeofday()` aus sys/time.h abgebildet wird.

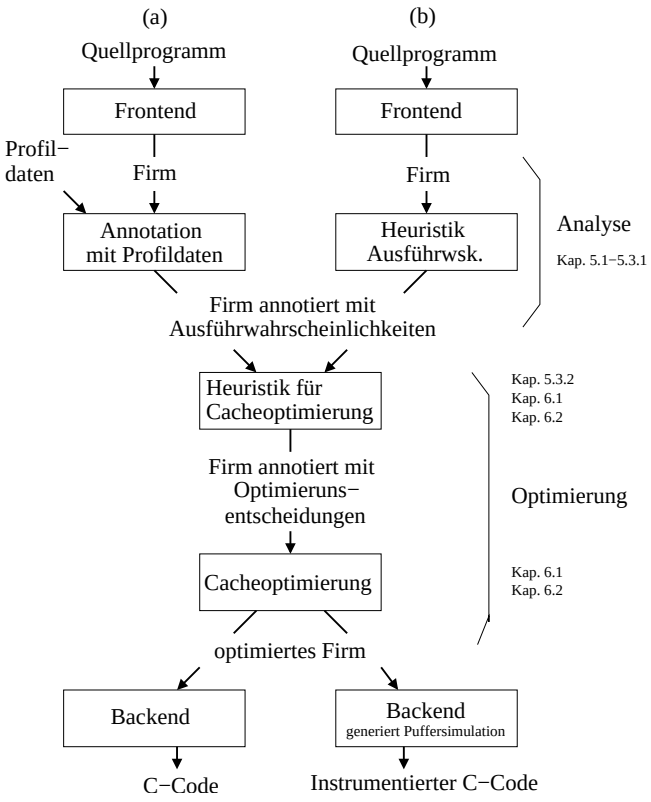

Abbildung 7.1: Pufferoptimierungen im jack-Übersetzer. (a) zeigt das Vorgehen bei Optimierung mit Profildaten, (b) bei Verwendung der Heuristik. Die Befehlserzeugung kann bei Bedarf Instrumentation mit einer Puffersimulation erzeugen.

Vergleich mit Optimierung anhand Profildaten

Um die Qualität unserer Heuristik zu messen, führen wir zusätzlich Experimente mit Profildaten durch. Vergleichbare Arbeiten aus der Literatur verwenden ausschließlich Profildaten. Die Profildaten erzeugen wir mit der oben erwähnten Simulation und einem unoptimierten Programm. Dabei verwenden wir kleinere Eingaben als für die eigentlichen Messungen. Die Profildaten entsprechen genau den in Kap. 5.3.1 abgeleiteten Werten. Die Profildaten ersetzen also die Heuristiken, die abschätzen welche Klassen bzw. Felder wie häufig verwendet wurden. Abb. 7.1 vergleicht den Ablauf einer Übersetzung mit Profildaten (links) im Vergleich mit einer Übersetzung mit unserer Heuristik (rechts).

Damit können wir unsere Heuristik zur Abschätzung pufferkritischer Verbunde validieren. Weiter führen wir unsere Optimierung mit diesen Daten durch. Können wir mit den Profildaten durch die Optimierung Performanzgewinne realisieren, beweist dies, dass die Optimierung funktioniert. Durch den Vergleich der Performanzgewinne auf Basis von Profildaten und heuristischen Daten können wir die Qualität der heuristischen Daten beurteilen.

7.2 Bewertung der Analyse

In diesem Abschnitt bewerten wir die Ergebnisse unserer Analyse. Zunächst validieren wir einige Annahmen, die wir in unserer Analyse treffen, durch Messungen. Dann untersuchen wir, ob die Analyse die Felder und Verbundtypen als pufferkritisch einstuft, die auch bei einer Simulation als pufferkritisch auffallen. In einem weiteren Schritt vergleichen wir die Optimierungsentscheidungen auf Basis der Simulationsdaten mit den Optimierungsentscheidungen auf Basis der Analyse.

7.2.1 Ausführungshäufigkeiten von Grundblöcken und Methoden

Die Heuristik zur Bewertung der Ausführungshäufigkeiten von Grundblöcken in Kap. 5.1 macht verschiedene Annahmen über bedingte Sprünge. Hier wollen wir untersuchen, ob diese Annahmen berechtigt sind, oder ob für bestimmte Sprünge genauere Vorhersagen getroffen werden können.

Tabelle 7.2: Bedingte Sprünge in den Olden Testprogrammen

Messung	BH	Bisort	Em3d	Health	MST
Nullvergleich wahr	17027	12434945	254	2053427	499637
Nullvergleich falsch	14575	12731594	260583	2028280	2089758
Ausnahme eingetreten	0	0	0	0	0
keine Ausnahme	2678421	46596055	11653199	22848860	19938653
Raise erreicht	0	0	0	0	0
Raise passiert	2977717	3428	11843956	8568489	7498181
Messung	Perimeter	Power	TreeAdd	TSP	Voronoi
Nullvergleich wahr	8196093	1604409	125	218572	642182
Nullvergleich falsch	8224059	1224679	479	175182	364980
Ausnahme eingetreten	0	0	0	0	0
keine Ausnahme	15719285	156376175	12585406	51571265	29094543
Raise erreicht	0	0	0	0	0
Raise passiert	3407	213488209	2100356	2723	7925967

Nullvergleiche

Nullvergleiche dienen häufig dem Abbruch einer Schleife. Der Wert Null verlangt in vielen Fällen eine Ausnahmebehandlung, da er zur Markierung illegaler Zustände oder uninitialisierter Variablen verwendet wird. Manche Operationen sind für Null illegal. Daher stellt sich die Frage, ob Vergleiche mit Null häufig fehlschlagen. In diesem Fall könnte die Heuristik bei bedingten Sprüngen nach Nullvergleichen das Sprungziel im Fall ungleich Null als häufiger erreicht markieren.

Tabelle 7.2 zeigt eine Messung der Nullvergleiche in den Olden Testprogrammen. Um diese Zahlen zu generieren haben wir den Codegenerator so geändert, dass er in Grundblöcken hinter Nullvergleichen Zählererhöhungen einfügt. Vorher haben wir im Steuerfluss kritische Kanten entfernt, so dass für jedes Vergleichsergebnis ein eindeutiger Block vorhanden ist.

An Tabelle 7.2 lässt sich leicht erkennen, das es keine klare Tendenz für Nullvergleiche gibt.

Ausnahmen

Die Zwischensprache Firm unterscheidet zwei Arten von Ausnahmen. Die einen werden direkt von Operationen der Zwischensprache erzeugt, deren Semantik eine Ausnahme vorsieht. Die anderen werden durch eine explizite Operation der Zwischensprache, Raise, dargestellt. Mit diesem Operator werden durch die Programmiersprache vorgesehene Ausnahmen ausgedrückt, die nicht direkt mit einer Ausnahme der Zielmaschine korrespondieren.

Wir messen das Auftreten beider Arten von Ausnahmen. Dabei platzieren wir, wie beim Messen der Nullvergleiche,in beiden Steuerflussästen nach der Entscheidung über Ausnahmen einen Zähler. Bei Ausnahmen durch die Raise-Operation ist dies nur möglich, wenn sich in der Methode, in der sich die Raise-Operation befindet, vorher ein bedingter Sprung befindet. Dies ist bis auf eine Ausnahme (`get()` in `java/util/Collections$EmptyList`) immer der Fall. Aus messtechnischen Gründen werden bei der Java Ausnahme `IndexOutOfBound` nicht genommene Ausnahmen doppelt gezählt.

Tabelle 7.2 zeigt, dass in den Olden Testprogrammen keine Ausnahmen auftreten. Die Annahme einer geringen Wahrscheinlichkeit für Verzweigungen, die zu Ausnahmen führen, ist also gerechtfertigt.

Präzision des Aufrufgraphen

Die Präzision des Aufrufgraphen entscheidet darüber, wie genau Rekursionen abgeschätzt werden können. Durch Optimierung der Aufrufstellen und Offenen Einbau kann der Aufrufgraph eingeschränkt werden.

Tabelle 7.3 zeigt, wie viele Aufrufstellen welcher Art in den Testprogrammen vorkommen. Dabei werden Aufrufe an eine statisch bekannte Methode und Aufrufe über dynamische Methodenselektion unterschieden. In jeder Spalte finden sich zwei Werte. Der erste berücksichtigt nur Aufrufstellen in den Methoden der Testprogramme. Der zweite berücksichtigt auch Aufrufstellen in der API. Durch entfernen unbenutzter Methoden der API durch den Übersetzer unterscheiden sich diese für die verschiedenen Testprogramme.

Die erste Spalte gibt den Namen des Testprogramms an. Die zweite Spalte (#opt poly) gibt an, wie viele dynamische Aufrufstellen zu statischen optimiert wurden. Die dritte Spalte (statisch) nennt die Anzahl statischer Aufrufe an Methoden in der Übersetzungseinheit, die vierte (extern) ebensolche an unbekannte Methoden, das heißt C Implementierungen des Laufzeitsystems

Tabelle 7.3: Aufrufstellen und deren Optimierung. Paare x/y bedeuten: x Aufrufstellen im Testprogramm, y Aufrufstellen in Testprogramm und API.

Programm	#opt poly	statisch	extern	dynamisch	Ziele
	ohne/mit API				
BH	130 / 222	252 / 470	22 / 40	19 / 94	38 / 498
Bisort	44 / 138	104 / 324	7 / 25	1 / 75	15 / 439
Em3d	92 / 186	177 / 395	6 / 25	0 / 75	0 / 442
Health	79 / 173	142 / 365	5 / 25	0 / 75	0 / 434
MST	63 / 153	102 / 317	5 / 23	2 / 76	30 / 454
Perimeter	29 / 119	97 / 311	7 / 25	33 / 107	87 / 519
Power	60 / 146	168 / 367	19 / 37	0 / 74	0 / 418
TreeAdd	26 / 116	87 / 301	5 / 23	0 / 74	0 / 424
TSP	63 / 156	130 / 346	12 / 31	0 / 74	0 / 424
Voronoi	215 / 311	253 / 501	11 / 30	57 / 197	114 / 1025

wie `arraycopy`. Die fünfte Spalte (dynamisch) gibt die Anzahl dynamischer Aufrufstellen nach der Optimierung an. Die letzte Spalte (Ziele) zeigt, wie viele verschiedene Methoden über die dynamischen Aufrufstellen aufgerufen werden können.

Es zeigt sich, dass selbst mit den einfachen Optimierungen in jack ein Großteil der dynamischen Aufrufstellen zu statischen reduziert werden kann. Der Anteil dynamischer Aufrufstellen an allen Aufrufstellen ist recht gering, durch im Schnitt 5-10 mögliche Ziele tragen diese Aufrufstellen jedoch deutlich zur Komplexität des Aufrufgraphen bei. Polymorphie findet sich insbesondere in BH, MST, Perimeter und Voronoi.

7.2.2 Bewertung der Identifikation pufferkritischer Verbunde

Hier vergleichen wir die Heuristik zum Auffinden pufferkritischer Verbunde mit Ergebnissen der Simulation. Dies zeigt, ob die Heuristik korrekt Verbunde erkennt, die viele Fehlzugriffe erzeugen. Damit weisen wir nach, ob unsere Heuristik sinnvoll als statische Analyse für Pufferoptimierungen einsetzbar ist, sei es unsere oder Optimierungen aus der Literatur.

In diesem Kapitel diskutieren wir Daten für einzelne Java Klassen und

für einzelne Felder aus diesen Klassen. Da die Namen der Java Klassen aus der API sehr lang sind, kürzen wir diese ab. Das Kürzel j/ steht für einen Pfad zu einer API Klasse, der mit ‚java' beginnt, wie zum Beispiel java/util/. Weiter kürzen wir WeakHashMap mit WHM, Hashtable mit HT und EncoderEightBitLookup mit EEBL ab. Ansonsten richten sich die Namen nach Java Konventionen. [Lj/WHM$WeakBucket; bezeichnet also eine Reihung (Präfix [), die Referenzen (L<Klasse>;) auf Objekte vom Typ WeakBucket enthält, der eine innere Klasse ($) von java/util/WeakHashMap ist.

In diesem Abschnitt vergleichen wir gemessene und heuristisch geschätzte Häufigkeiten von Allokationen und Feldzugriffen. Dazu haben wir exemplarisch zwei Testprogramme ausgewählt, Perimeter und BH. Bei Perimeter werden mit heuristischen und gemessenen Daten die gleichen Optimierungsentscheidungen getroffen, bei BH unterschiedliche.

Allokationshäufigkeiten

Allokationshäufigkeiten sind ein Maß für den Speicherbedarf der Objekte eines Typs.

Die Tabellen 7.4 und 7.5 zeigen Allokationshäufigkeiten. Die ersten drei Spalten geben gemessene Allokationshäufigkeiten an, die drei nächsten Spalten mit der Heuristik aus Kap. 5.3.2 geschätzte. Die Einträge sind jeweils nach ihrer Häufigkeit sortiert. Die doppelte Linie stellt den nach Kap. 5.3.4 ermittelten Schwellenwert dar. Dessen Bedeutung werden wir in Kap. 7.2.3 diskutieren.

In der Heuristik betrachten wir nur Klassen, die in Schleifen oder Rekursionen alloziert werden. Daher ist hier die Liste kürzer. Die Messung berücksichtigt nur Allokationen, die bei der Programmausführung erreicht werden. Die Heuristik kann nicht alle Allokationen, die praktisch nie erreicht werden, erkennen. Daher können hier Klassen vorkommen, die bei der Messung nicht erscheinen.

Die Tabellen nennen in Spalten zwei und fünf absolute Zahlen, in Spalten drei und sechs auf den am häufigsten allozierten Typ normierte Zahlen. Die absoluten Werte hängen bei der Messung von den Eingaben der Testprogramme ab. Bei der Heuristik bestimmen frei gewählte Konstanten die absoluten Werte. Absolute Zahlen sind für die Optimierung nicht von Belang. Die Optimierung interessiert nur, welche Typen häufiger alloziert werden als andere, und welche häufiger zugegriffen werden, also Teilmengen aller Typen und nicht einzelne Typen. Wenn diese Teilmengen bei beiden Techniken die

Tabelle 7.4: Gemessene und geschätzte Allokationshäufigkeit für BH. Die doppelte Linie kennzeichnet den durch die Heuristik festgelegten Schwellenwert für pufferkritische Klassen.

BH					
Gemessene Allokationen			Geschätzte Allokationen		
MathVector	743282	1.0000	[C	6921.04	1.0000
[D	743282	1.0000	MathVector	3738.94	0.5402
Node$HG	10000	0.0135	[D	3716.94	0.5370
Cell	5194	0.0070	j/AbstractList$1	3667.81	0.5300
[LNode;	5194	0.0070	j/HT$HashIterator	3284.37	0.4745
Body2Enumerate	300	0.0004	j/String	3197.29	0.4620
Body	101	0.0001	j/WHM$2	2150.34	0.3107
[C	26	0.0000	Cell	1079.98	0.1560
[B	17	0.0000	[LNode;	1079.98	0.1560
j/String	12	0.0000	[B	711.75	0.1028
j/Integer	2	0.0000	j/StringBuffer	676.99	0.0978
j/PrintStream	2	0.0000	Node$HG	100.00	0.0144
j/PrintWriter	2	0.0000	Body2Enumerate	30.00	0.0043
j/OutputStreamWriter	2	0.0000	Body	10.00	0.0014
gnu/j/Encoder8859_11	2	0.0000	j/Integer	7.50	0.0011
j/FileOutputStream	2	0.0000			
j/BufferedOutputStream	2	0.0000			
[Lj/String;	1	0.0000			
Tree	1	0.0000			
j/ReferenceQueue	1	0.0000			
j/WHM$WeakEntrySet	1	0.0000			
[Lj/WHM$WeakBucket;	1	0.0000			
j/BufferedInputStream	1	0.0000			
j/FileInputStream	1	0.0000			
[Lj/HT$HashEntry;	1	0.0000			
Body1Enumerate	1	0.0000			

Tabelle 7.5: Gemessene und geschätzte Allokationshäufigkeit für Perimeter. Die doppelte Linie kennzeichnet den durch die Heuristik festgelegten Schwellenwert für pufferkritische Klassen.

Perimeter					
Gemessene Allokationen			Geschätzte Allokationen		
BlackNode	170064	1.0000	j/HT$HashIterator	3350.50	1.0000
WhiteNode	169624	0.9974	[C	3106.26	0.9271
GreyNode	113229	0.6658	j/AbstractList$1	3076.58	0.9182
[C	31	0.0002	j/WHM$2	1854.23	0.5534
[B	21	0.0001	j/String	1773.07	0.5292
j/String	16	0.0001	[B	701.75	0.2094
j/PrintStream	2	0.0000	j/StringBuffer	312.50	0.0933
j/PrintWriter	2	0.0000	j/Integer	5.00	0.0015
j/OutputStreamWriter	2	0.0000	WhiteNode	3.00	0.0009
gnu/j/Encoder8859_11	2	0.0000	BlackNode	2.25	0.0007
j/FileOutputStream	2	0.0000	GreyNode	0.75	0.0002
j/BufferedOutputStream	2	0.0000			
[Lj/String;	1	0.0000			
j/Integer	1	0.0000			
j/ReferenceQueue	1	0.0000			
j/WHM$WeakEntrySet	1	0.0000			
[Lj/WHM$WeakBucket;	1	0.0000			
j/BufferedInputStream	1	0.0000			
j/FileInputStream	1	0.0000			
[Lj/HT$HashEntry;	1	0.0000			

entscheidenden Typen enthalten ist eine ausreichende Übereinstimmung gegeben. Daher diskutieren wir die Reihenfolge der Datentypen in den Tabellen, sowie den Abstand zwischen benachbarten Werten.

Bei BH stimmen die geschätzten Allokationshäufigkeiten gut mit denen der Messung überein. Die zwei am häufigsten allozierten Klassen, MathVector und [D, sind auch bei der Heuristik weit oben. Die für die Optimierung relevanten Klassen, Cell und [Node, sind bei der Heuristik ebenfalls weit oben. Bei der Heuristik liegen die einzelnen Werte recht nah beieinander. So ist der Wert für die Datenstruktur in mittlerer Position (Cell bei BH, [B bei Perimeter) nur ca. 1/5 des obersten Wertes. Die gemessenen Werte fallen deutlich schneller. Dadurch ist es anhand der heuristischen Daten schwerer, einen Unterschied zwischen der Größe der einzelnen Datenstrukturen zu erkennen.

Bei den Allokationshäufigkeiten für Perimeter verschätzt sich die Heuristik komplett. In Perimeter wird ein Baum alloziert, der aus Knoten der drei Klassen WhiteNode, BlackNode und GreyNode besteht. Dies passiert in einer einzigen Rekursion, die dazu noch abhängig von einer Bedingung einen der drei Knoten alloziert. Die rekursive Methode ruft sich vier mal selbst auf, um einen Vierfachbaum aufzubauen.

Die Rekursion wird durch die Heuristik, wie alle Schleifen und Rekursionen, mit einem konstanten Faktor bewertet. Die bedingten Verzweigungen in der rekursiven Methode verringern diesen Faktor. Die Heuristik erkennt nicht, dass hier eine exponentiell wachsende Rekursion vorliegt, da die Methode eine einzige Zusammenhangskomponente darstellt, die als ganzes bewertet wird. Daher ergeben sich für diese Knoten zu geringe Allokationshäufigkeiten.

Zugriffshäufigkeiten

Mit Zugriffshäufigkeiten teilen wir Felder in pufferkritische und unkritische ein. Kombiniert mit Allokationshäufigkeiten sind sie ein Maß für Wiederverwendung.

Tabelle 7.6 zeigt akkumulierte Zugriffshäufigkeiten für die Klassen Cell und Body aus BH. Der gemeinsame Obertyp Node wird nie alloziert, daher betrachten wir diese Klasse nicht gesondert. Felder mit Suffix `inh` sind von Node geerbt. Da wir akkumulierte Häufigkeiten (Kap. 5.3.3) betrachten, sind Zugriffe auf Felder über den Datentyp Node auf die eingeerbten Felder in Body und Cell verteilt. Die Referenz auf die Sprungtabelle ist mit `disp` bezeichnet. In vielen Fällen ist der Wert für dieses Feld eins. In diesen Fällen

Tabelle 7.6: Gemessene und geschätzte akkumulierte Zugriffshäufigkeit für Felder von Cell und Body aus BH. Die doppelte Linie kennzeichnet den durch die Heuristik festgelegten Schwellenwert für pufferkritische Felder.

Gemessene Zugriffe			Geschätzte Zugriffe		
BH – Cell					
subp	212.24	1.0000	pos_inh	14.76	1.0000
pos_inh	136.67	0.6439	subp	2.91	0.1972
mass_inh	69.99	0.3298	mass_inh	1.77	0.1199
next	1.00	0.0047	disp	1.14	0.0772
disp	1.00	0.0047	next	1.00	0.0678
BH – Body					
vel	404.92	1.0000	disp	1393.49	1.0000
procNext	299.02	0.7385	vel	44.10	0.0316
newAcc	199.02	0.4915	procNext	32.10	0.0230
acc	198.03	0.4891	acc	26.10	0.0187
pos_inh	136.67	0.3375	newAcc	21.10	0.0151
phi	100.01	0.2470	pos_inh	14.76	0.0106
mass_inh	69.99	0.1728	phi	11.10	0.0080
next	3.00	0.0074	next	3.60	0.0026
disp	1.00	0.0025	mass_inh	1.77	0.0013

wird kein dynamischer Aufruf ausgeführt, der einzige Zugriff auf dieses Feld ist die Initialisierung.

In beiden Fällen bringt die Heuristik die Felder in eine den gemessenen Daten sehr ähnliche Reihenfolge. Allerdings sind die Abstufungen der Werte sehr unterschiedlich. Die Heuristik verschätzt sich bei den Zugriffen auf die Sprungtabelle bei Body stark. Dies ergibt sich durch eine Inferenz der Heuristik mit offenem Einbau. Da Body und Cell Blätter in der Vererbungshierarchie sind, kann in Methoden dieser Datentypen Polymorphie leicht aufgelöst werden. Daraufhin können geerbte Methoden offen eingebaut werden, worauf hin wiederum polymorphe Aufrufe aufgelöst werden. Trotzdem bleiben die ursprünglichen, nun offen eingebauten Methoden für den Obertyp bestehen. Diese enthalten nach wie vor polymorphe Aufrufe, die als Zugriffe auf die Sprungtabelle des Obertypen gelten. Durch die Akkumulation der Häufigkeiten werden diese auch den Untertypen zugeschlagen.

Tabelle 7.7: Gemessene und geschätzte akkumulierte Zugriffshäufigkeit für Felder von BlackNode aus Perimeter. Die doppelte Linie kennzeichnet den durch die Heuristik festgelegten Schwellenwert für pufferkritische Felder.

Gemessene Zugriffe			Geschätzte Zugriffe		
Perimeter – BlackNode					
parent_inh	7.04	1.0000	quadrant_inh	31.00	1.0000
quadrant_inh	6.41	0.9105	parent_inh	16.00	0.5161
nw_inh	3.47	0.4929	nw_inh	11.85	0.3823
ne_inh	3.22	0.4574	ne_inh	11.35	0.3661
sw_inh	3.22	0.4574	sw_inh	11.35	0.3661
se_inh	3.22	0.4574	se_inh	11.35	0.3661
disp	1.00	0.1420	disp	1.00	0.0323

Tabelle 7.7 zeigt die akkumulierten Zugriffshäufigkeiten für BlackNode aus Perimeter. Die Werte für GreyNode und WhiteNode sind diesen vergleichbar. Heuristik und Messung finden die gleiche Reihenfolge, bis auf direkte Vertauschungen benachbarter Felder. Die Heuristik kann also sehr gut die realen Zugriffshäufigkeiten nachbilden.

Zusammenfassend lässt sich sagen, dass die Heuristik recht gut Allokations- und Zugriffshäufigkeiten einschätzen kann. Häufig kommt es zu einer sehr ähnlichen, relativen Bewertung. Es gibt jedoch auch Fälle, in denen die Heuristik eine ungünstige Reihenfolge findet oder aber paarweise Relationen schlecht einschätzt.

7.2.3 Bewertung der Optimierungsentscheidungen

Hier vergleichen wir die Optimierungsentscheidungen anhand der gemessenen und mit der Heuristik ermittelten Werte. Zunächst betrachten wir die Auswahl von Typen für Typanhäufen.

Entscheidung für Typanhäufen

Tabelle 7.8 zeigt die Anzahl der Klassen, die bei den einzelnen Olden Programmen für Typanhäufen ausgewählt wurden. Die erste Spalte gibt den Namen der Testprogramme an. Bei der Angabe von Klassen unterscheiden

wir jeweils Klassen der Testprogramme (Prog.) und Klassen aus der Java API. Die Spalten mit Klassen der Testprogramme berücksichtigen jeweils auch Reihungsklassen, die Referenzen auf Klassen aus den Testprogrammen enthalten. Wir ordnen diese vom Übersetzer generierten Klassen den Testprogrammen zu, da diese Reihungen in diesen definiert sind und nicht direkt in der API verwendet werden können.

Die zweite und dritte Spalte geben die Anzahl übersetzter Klassen an. Die vierte und fünfte Spalte geben die Anzahl der Klassen an, die optimiert werden, wenn wir Profildaten für die Optimierung verwenden. Spalten sechs und sieben bewerten die Entscheidung anhand der Heuristik.

In Spalten sechs und sieben finden sich je zwei Zahlen. Zuerst die Anzahl der Klassen, die mit der Optimierung anhand Profildaten übereinstimmen, sodann die Anzahl zusätzlich für die Optimierung ausgewählter Klassen.

Es zeigt sich, dass die Optimierung anhand Profilinformationen praktisch nur Klassen aus den Testprogrammen optimiert. Bei BH und Power wurden aus der API einige Klassen, die Ausgabefunktionalität bereitstellen, optimiert. Weiter zeigt sich, dass mit der Heuristik viele der Klassen aus den Testprogrammen optimiert werden, die auch durch die Profilinformationen als pufferkritisch eingestuft wurden. Bei Voronoi stuft die Heuristik einige Klassen der API als pufferkritisch ein. Dabei wird insbesondere die Klasse java/util/Collections$SynchronizedIterator als problematisch eingestuft. Diese enthält rekursive Methoden bei denen der rekursive Aufruf polymorph ist. Die Heuristik erkennt, dass hier intensive Berechnungen stattfinden können, jedoch nicht, dass dies im konkreten Fall nicht passieren wird, da die Methode nie erreicht wird.

Insgesamt zeigt sich, dass die Heuristik konservativer ist, bis auf wenige Fälle jedoch die gleichen Klassen zur Optimierung vorschlägt wie die Analyse mit Profildaten. In beiden Fällen werden Klassen als pufferkritisch eingestuft, die durch reines Betrachten als nicht relevant erkannt werden können, zum Beispiel die API Klassen.

An den Beispielen BH und Perimeter betrachten wir die Entscheidungsfindung für Typanhäufen etwas detaillierter.

Die Optimierung anhand der gemessenen Daten versucht für BH zunächst, in den anhand der Allokationshäufigkeit (Siehe Tabelle 7.4) pufferkritisch eingestuften Klassen geeignete Haufen zu finden. Da aber mit MathVector, [D und Node$HG keine Haufen gebildet werden können, werden alle Datentypen als Knoten im Haufengraph herangezogen. Dadurch wählt die Heuristik anhand der Feldzugriffhäufigkeiten aus, da diese die Kanten

Tabelle 7.8: Übereinstimmung der Optimierungsentscheidung für Typanhäufen anhand heuristischer und gemessener Daten für die Olden Testprogramme

	Anz. übers. Klassen		Anzahl optimierter Klassen			
			Profildaten		Heuristik	
Name	Prog.	API	Prog.	API	Prog.	API
BH	9+1	184	3	5	1 1	- -
BiSort	2	181	1	-	1 -	- -
Em3d	4+1	181	2	-	2 -	- -
Health	8+3	181	7	-	2 -	- -
MST	6+2	181	1	-	1 4	- -
Perimeter	10	181	3	-	3 -	- -
Power	6+2	181	2	5	2 -	- -
TreeAdd	2	181	1	-	1 -	- -
TSP	2	181	1	-	1 -	- -
Voronoi	6+1	188	2	-	2 1	- 5

im Graphen bestimmen. Da die Menge der Knoten im Haufengraph nicht eingeschränkt ist, werden relativ viele Haufen gefunden. Tabelle 7.9 zeigt die Haufen für BH. Die zweite Spalte (Profildaten→Haufen) nennt die gefundenen Haufen mit Profildaten, die sechste (Heuristik→Haufen) die Haufen mit heuristischen Daten. Jede Zahl steht für einen Haufen. Datentypen mit der gleichen Zahl werden im gleichen Haufen platziert.

Mit heuristischen Daten ist der Schwellenwert für die Allokationshäufigkeiten (Tabelle 7.4) deutlich weiter unten. Unter den Klassen oberhalb dieses Schwellenwertes findet die Heuristik einen Haufen, der die Klasse Cell enthält.

Bei Perimeter wurden anhand Profildaten drei Datentypen als häufig alloziert eingestuft (Tabelle 7.5). Diese bilden einen Haufen, der für Typanhäufen ausgewählt wird. Obwohl die Klassifikation häufig allozierter Datentypen mit heuristischen Daten völlig anders aussieht, wählt die Optimierung den gleichen Haufen für Typanhäufen aus. Da die Optimierung unter den häufig allozierten Klassen keinen geeigneten Haufen finden kann, werden auch hier alle Datentypen als Knoten für den Haufengraph betrachtet. Da die Kantenfelder von BlackNode, GreyNode und WhiteNode mit einer relativ hohen Zugriffshäufigkeit bewertet wurden (Tabelle 7.7), werden diese drei Datentypen als Haufen ausgewählt. Die Optimierung kommt also auf verschiedenen Wegen zu der gleichen Entscheidung.

Entscheidung für Verbundteilen

Wir betrachten die Entscheidungen für Verbundteilen wieder anhand der Olden Testprogramme BH und Perimeter. Tabelle 7.9 zeigt die Optimierungsentscheidungen für das Programm BH. Die erste Spalte zeigt die Namen optimierter Klassen. Die nächsten vier Spalten zeigen Optimierungsentscheidungen anhand Profildaten, die letzten vier Spalten zeigen Optimierungsentscheidungen anhand der Heuristik. Die erste Spalte jedes Viererblocks zeigt die Nummer des zugeordneten Typhaufens. Die nächsten drei Spalten zeigen wie eine Klasse geteilt wurde. Diese drei Spalten unterscheiden sich in der gewählten Variante von Verbundteilen. Die Spalte ‚Optimal' zeigt die Optimierungsentscheidungen bei optimalem Verbundteilen, die Spalte ‚Adaptiv' bei adaptivem, typsicheren Verbundteilen, und die dritte Spalte (‚Komb.') eine Variante, die die beiden vorhergehenden Verfahren kombiniert und versucht, möglichst wenig Datentypen zu teilen, die nicht in einem Haufen platziert werden. Die Heuristik für die Spalten ‚Optimal' und ‚Komb.' betrachtet auch inkompatibles, optimales Verbundteilen. Dies konnte bei BH und Perimeter nicht angewandt werden, kam jedoch bei Health, Power und TSP zum Tragen.

Der Eintrag *op* steht für optimales Verbundteilen, das heißt, der abgespaltene Teil enthält ein Feld mit einer Rückreferenz, der pufferkritische Teil enthält keine Referenz auf die Sprungtabelle. Ist ein *op*-Eintrag mit einem Stern gekennzeichnet wurde dem pufferkritischen Teil zusätzlich eine Referenz auf die Sprungtabelle hinzugefügt. Dies ist der Fall wenn bei diesem Typ, im Gegensatz zu anderen Typen die aufgrund der Vererbung gemeinsam geteilt werden, häufig auf die Sprungtabelle zugegriffen wird. *ts* steht für adaptives, typsicheres Verbundteilen.

Die Datentypen Body, Cell, Node und [LNode; stammen aus dem Testprogramm, die anderen Datentypen aus der API. Nur Datentypen, die für Typanhäufen ausgewählt wurden, sind Kandidaten für Verbundteilen. Um Typkompatibilität in der Vererbungshierarchie zu gewährleisten müssen eventuell noch weitere Datentypen geteilt werden, wie zum Beispiel Node, der gemeinsame Obertyp von Body und Cell, der jedoch nie alloziert wird.

Mit Profildaten werden verschiedene Klassen mit Ein/Ausgabefunktionalität optimiert. Diese Klassen sind für die Ausgabe von 8 Zeilen Information über das Programm und das Parsen der Kommandozeile verantwortlich und stellen keine ernsthaft pufferkritische Datenstruktur dar.

Unabhängig von der Ausprägung der Optimierung werden die gleichen

Tabelle 7.9: Optimierungsentscheidungen für das Olden Testprogramm BH.

	Profildaten				Heuristik			
Klassenname	Haufen	Optimal	Adaptiv	Komb.	Haufen	Optimal	Adaptiv	Komb.
Body	1	op	ts	ts		op*	ts	op*
Cell	3	op	ts	ts	1	op	ts	op
Node		op	ts			op	ts	op
[LNode;	3							
gnu/j/Encoder		op	ts	op				
gnu/j/Encoder8859_11	4	op	ts	op				
gnu/j/EEBL		op	ts	op				
j/BufferedInputStream	2							
j/OutputStreamWriter	4	op	ts	op				
j/PrintWriter	4							
j/PrintStream	4	op	ts	op				

Tabelle 7.10: Optimierungsentscheidungen für das Olden Testprogramm BH für Body und Cell. ‚k‘: Feld in pufferkritischem Teil, ‚u‘: Feld in unkritischem Teil.

	Profildaten			Heuristik		
Feldname	Optimal	Adaptiv	Komb.	Optimal	Adaptiv	Komb.
BH – Body						
Body.pos_inh	k	k	k	k	k	k
Body.procNext	k	k	k	k	k	k
Body.newAcc	k	k	k	k	k	k
Body.acc	k	k	k	k	k	k
Body.phi	k	k	k	k	k	k
Body.vel	k	k	k	k	k	k
Body.splitref	k	k	k	k	k	k
Body.mass_inh	k	k	k	u	u	u
Body.disp	u	k	k	u+k	k	u+k
Body.next	u	u	u	u	u	u
Body.splitbackref	u	-	-	u	-	u
BH – Cell						
Cell.pos_inh	k	k	k	k	k	k
Cell.splitref	k	k	k	k	k	k
Cell.mass_inh	k	k	k	u	u	u
Cell.subp	k	k	k	u	u	u
Cell.disp	u	k	k	u	k	u
Cell.next	u	u	u	u	u	u
Cell.splitbackref	u	-	-	u	-	u

Datentypen geteilt. Nur bei Profildaten→Komb. wird die Klasse Node nicht geteilt. Dies ist jedoch unbedeutend, da die Klasse Node nie alloziert wird. Node ist Obertyp von Body und Cell und enthält die pufferkritischen Felder. Da Node eine abstrakte Klasse ist, kann es keine Instanzen von Node geben, Node ist also selbst nicht pufferkritisch. Node zu teilen bedeutet dann aber auch keinen unnötigen Zusatzaufwand.

Die Aufteilung von Body und Cell wird anhand der Daten in Tabelle 7.6 vorgenommen. Das Teilungsergebnis zeigt Tabelle 7.10. Die doppelte Trennlinie in Tabelle 7.6 gibt den Schwellenwert für pufferkritische Felder an. Die Heuristik stuft weniger Felder als pufferkritisch ein und kommt dadurch zu kleineren Teildatentypen, die in den Haufen platziert werden. Dafür sind mehr indirekte Zugriffe zu erwarten. Da die Heuristik bei Body die Zugriffe auf die Sprungtabelle über das Feld `disp` überschätzt, wird dieses Feld bei optimalem Verbundteilen im pufferkritischen Teil dupliziert. Dies vermeidet Indirektionen beim Methodenaufruf. Die Optimierung anhand Profildaten teilt die Klasse mit adaptivem, typsicheren Verbundteilen, obwohl im unkritischen Teil nur ein Feld platziert wird. Dies ergibt sich, da das Feld `disp` als unkritisch eingestuft wird, damit also zwei Felder abgespalten werden können. Durch den Teilungsalgorithmus wird aber dieses Feld in den pufferkritischen Teil gezwungen.

Die Zugriffshäufigkeiten bei Perimeter (Tabelle 7.7) sind für die verschiedenen Felder sehr ähnlich, so dass entweder alle Felder als unkritisch oder alle Felder bis auf `disp` als pufferkritisch eingestuft werden. Damit lohnt sich Verbundteilen nicht.

Die Optimierungsentscheidungen stimmen also in den entscheidenden Programmteilen weitgehend überein.

7.3 Bewertung der Optimierung

In diesem Kapitel wollen wir nun die Laufzeitverbesserungen durch die Optimierungen diskutieren.

Tabelle 7.11 zeigt die Laufzeiten der bereits vorgestellten Optimierungsvarianten relativ zur Laufzeit des unoptimierten Programms in Prozent. Negative Zahlen zeigen eine Laufzeitverbesserung an. Zum Beispiel ist BH bei Optimierung anhand Profildaten durch Anhäufen 5.08% schneller als das unoptimierte Programm. Werden der gleichen Optimierung heuristische Daten zugrunde gelegt, wird das Programm 1,07% schneller. Die unterste Zeile zeigt

Tabelle 7.11: Laufzeiten der Olden Testprogramme relativ zum unoptimierten Programm in Prozent. Schnitt aus 10 Messungen.

	Profildaten				Heuristik			
Name	Anhäufen	Optimal	Adaptiv	Komb.	Anhäufen	Optimal	Adaptiv	Komb.
BH	-5.08	0.72	-4.06	-4.56	-1.07	-2.17	-3.63	-2.21
BiSort	-1.41	-1.44	-1.47	-1.55	-1.60	-1.45	-1.50	-1.54
Em3d	1.06	1.02	1.21	1.30	1.01	0.94	1.28	1.28
Health	-1.17	-2.04	-1.46	-1.48	14.71	21.76	20.59	21.66
MST	5.45	5.48	5.44	5.34	-3.46	-31.17	-2.27	-31.08
Perimeter	-0.81	-1.07	-1.19	-0.78	-1.23	-0.93	-1.14	-0.97
Power	1.23	1.19	1.11	1.09	1.14	0.89	0.72	0.81
TreeAdd	0.13	0.25	0.30	0.25	-0.50	0.13	-0.40	0.28
TSP	-2.83	-5.36	-1.61	-5.32	-3.60	-4.45	0.22	-4.55
Voronoi	1.41	1.52	1.52	1.49	-0.99	-1.51	-1.33	-1.21
Schnitt	-0.20	0.03	-0.02	-0.42	0.44	-1.80	1.25	-1.75

das arithmetische Mittel der Laufzeitverbesserungen. Im Schnitt über alle Testprogramme können bis zu 1,8% Laufzeitverbessserungen erreicht werden. In Einzelfällen beobachten wir bis zu 31,17% Laufzeitverbesserung aber auch 21,76% Laufzeitverschlechterung.

Die Tabelle zeigt, dass sowohl mit Profildaten wie auch mit heuristischen Daten deutliche Optimierungseffekte erreicht werden können. Weiter zeigt sich, dass diese Ergebnisse in beiden Fällen nicht konsistent erreicht werden können. So wird für BH bei optimalem Typteilen mit Profildaten kein Laufzeitgewinn erreicht, obwohl sich das Ergebnis der Optimierung nur geringfügig von den anderen Varianten unterscheidet.

Insbesondere werden MST mit Profildaten sowie Health mit heuristischen Daten bedeutend langsamer. Andererseits kann bei TSP mit Profil- und heuristischen Daten sowie bei MST mit heuristischen Daten ein sehr starker positiver Effekt erreicht werden.

Ähnliche Effekte sieht man in Tabelle 7.12, die die Änderung der Fehlzugriffsrate durch die Optimierung in Prozent zeigt. Da durch die Opti-

Tabelle 7.12: Änderung der Fehlzugriffsrate durch die Optimierung in Prozent.

	Profildaten				Heuristik			
Name	Anhäufen	Optimal	Adaptiv	Komb.	Anhäufen	Optimal	Adaptiv	Komb.
BH	-8.52	19.89	-7.39	-7.95	-5.68	11.36	10.23	11.36
BiSort	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Em3d	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.96
Health	9.87	8.35	9.07	8.67	40.37	44.70	45.10	44.70
MST	11.82	11.82	11.82	11.82	-3.65	-35.33	-3.36	-35.33
Perimeter	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Power	0.00	0.00	0.00	0.00	0.00	5.26	5.26	5.26
TreeAdd	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TSP	-42.21	-43.28	-41.81	-43.28	-43.85	-47.94	-42.09	-47.77
Voronoi	0.00	0.00	0.00	0.00	0.00	4.84	1.61	4.84

mierungen die Anzahl der Zugriffe steigen kann, berechnen wir die Fehlzugriffsrate bezüglich der Zugriffe im unoptimierten Programm. Die Tabellen mit der Laufzeit (Tabelle 7.11) und der Fehlzugriffsrate (Tabelle 7.12) sind nicht direkt vergleichbar. Die Fehlzugriffsrate wurde nur für den obersten Puffer ermittelt. Die Simulation verwendet virtuelle und nicht physische Adressen. Die simulierten Programme benötigen für die Simulation selbst zusätzlichen Speicher. Dadurch kann sich die Adressabbildung und damit das Pufferverhalten ändern.

Bei BH vermeidet zum Beispiel Anhäufen mit Profildaten 8,52% der Fehlzugriffe. Es gibt jedoch auch Fälle, in denen das Programm durch Optimierung schneller wird, obwohl mehr Fehlzugriffe auftreten (BH bei allen 3 Varianten von Verbundteilen mit Profildaten). Anhäufen kann neben der Pufferleistung auch Vorteile durch verringerte Seitenwechsel mit sich bringen. Bei BH ist die absolute Fehlzugriffsrate sehr gering (1,76%, siehe Tabelle 7.1). Typanhäufen trennt jedoch die häufig allozierte, kurzlebige Klasse MathVector von der langlebigen Datenstruktur. Daher kann die Veränderung der Rate um absolut ca. 0,2% eine geringere Bedeutung haben als eine dadurch erreichte Verbesserung der Seitenwechsel.

Bei Health wird mit der Heuristik ein deutlich kleinerer Haufen gebildet als mit Profildaten (Tabelle 7.8). Dieser Haufen enthält die programmeigene Listenklasse. Dadurch wird die Listenklasse von den Verbunden, die diese referenzieren, getrennt. Da für die Iteration jedoch ein Zugriff auf diese Klasse nötig ist, werden hier häufigere Fehlzugriffe verursacht. Da Health bereits im unoptimierten Programm eine hohe Fehlzugriffsrate hat, macht sich dies in der Laufzeit deutlich bemerkbar. Mit Profildaten wird die Listenklasse zusammen mit den Verbunden, die sie referenzieren, in einem Haufen platziert. Health zeigt optimiert mit Profildaten eine Laufzeitverbesserung, obwohl die simulierte Fehlzugriffsrate schlechter geworden ist. Dies kann an positiven Effekten in Speichern liegen, die durch die Simulation nicht erfasst werden.

Bei MST war ein großer Effekt zu erwarten, da MST unoptimiert die schlechteste Trefferrate hat. Mit Profildaten bildet die Optimierung einen Haufen, der nur die Klasse HashEntry enthält, für die nie Verbundteilen angewendet wird. Dadurch werden, ähnlich wie bei Health, sich referenzierende Verbunde getrennt. Das resultiert in zusätzlichen Fehlzugriffen. Mit der Heuristik werden genau diese Verbunde in den Haufen aufgenommen. Den entscheidenden Effekt bringt optimales Verbundteilen, das die Größe der Klasse Vertex um 3 Felder bzw. 60% verringert. Die Laufzeitdaten spiegeln bei MST die simulierten Fehlzugriffsraten wieder.

Bei TSP wird bei allen Optimierungen die Klasse Tree für Typanhäufen ausgewählt. Die Optimierungen unterscheiden sich jedoch im Ausmaß an Verbundteilen (Tabelle 7.13). Bei adaptivem Verbundteilen mit heuristischen Daten wird das Feld ‚disp' als unkritisch eingestuft, die Teilungsstrategie zwingt es jedong in den kritischen Teil. Dadurch verbleibt ein zu kleiner unkritischer Teil, die Optimierung sollte den Verbund nicht teilen. Das Programm wird dadurch nicht wie die anderen Optimierungsvarianten schneller. Dieser Messpunkt zeigt damit den negativen Effekt des Zusatzaufwandes durch Verbundteilen auf. Wir haben die Optimierung diesbezüglich verbessert.

Bei Voronoi werden mit heuristischen Daten mehr Haufen gebildet und zusätzlich optimales Verbundteilen angewandt. Die Optimierung erkennt, dass keine Typkompatibilität notwendig ist und erzeugt daher im unkritischen Teil keine Rückreferenzen.

Bei BiSort, Em3d und TreeAdd werden mit Heuristik und Profildaten die gleichen Haufen gebildet, Verbundteilen wird nicht angewandt. Die Problemgröße von Power kann nicht durch Programmparameter angepasst werden. Daher ist der Umfang von Power zu gering um mit der Pufferoptimierung ernsthafte Effekte zu erreichen.

Tabelle 7.13: Optimierungsentscheidungen für das Olden Testprogramm TSP. ‚k': Feld in pufferkritischen Teil, ‚u': Feld in unkritischen Teil.

	Profildaten			Heuristik		
Feldname	Optimal	Adaptiv	Komb.	Optimal	Adaptiv	Komb.
TSP – Tree						
Tree.next	k	k	k	k	k	k
Tree.x	k	k	k	k	k	k
Tree.y	k	k	k	k	k	k
Tree.splitref	k	k	k	k	k	k
Tree.left	u	u	u	k	k	k
Tree.right	u	u	u	k	k	k
Tree.prev	u	u	u	k	k	k
Tree.disp	u	k	u	u	k	u
Tree.sz	u	u	u	u	u	u

Tabelle 7.14: Laufzeiten der Olden Testprogramme mit zusätzlichen Feldern relativ zum unoptimierten Programm in Prozent. Schnitt aus 10 Messungen.

	Profildaten				Heuristik			
Name	Anhäufen	Optimal	Adaptiv	Komb.	Anhäufen	Optimal	Adaptiv	Komb.
Em3d	1.06	1.02	1.21	1.30	1.01	0.94	1.28	1.28
Em3dLarge	1.92	-2.49	-0.55	-2.63	2.27	-2.26	-0.45	-2.55
TSP	-2.83	-5.36	-1.61	-5.32	-3.60	-4.45	0.22	-4.55
TSPLarge	-1.13	-8.42	-5.06	-8.50	-1.88	-8.55	-7.67	-8.33

7.3.1 Effekt von Verbundteilen

Da die Olden Testprogramme hauptsächlich dynamische Datenstrukturen demonstrieren, jedoch diese nicht wirklich verwenden, finden sich in diesen Datenstrukturen wenig unkritische Felder. Um den Effekt von Verbundteilen genauer zu untersuchen, haben wir die Datenstrukturen in zwei der Olden Testprogramme um weitere, unbenutzte Felder ergänzt. Bei Em3d haben wir dem Datentyp Node Felder mit insgesamt 27 Byte hinzugefügt, bei TSP haben wir Tree um 27 Byte vergrößert. Die dadurch erhaltenen Programme nennen wir Em3dLarge und TSPLarge. Tabelle 7.14 zeigt die Laufzeitverbesserungen dieser Programme im Vergleich mit den Verbesserungen der normalen Olden Testprogramme.

Es zeigt sich, dass bei den großen Programmen Verbundteilen deutlich mehr Effekte zeigt. So können beide große Programme von Anhäufen allein nicht profitieren, gewinnen aber immer, wenn zusätzlich Verbundteilen angewandt wird. Bei Em3d zeigt die Optimierung überhaupt erst bei dem großen Programm Wirkung. Damit verdeutlicht dieses Experiment den Effekt von Verbundteilen.

Da die hinzugefügten Felder im Programm nicht benutzt werden, ist es für die Heuristik sehr einfach, sie als unkritisch einzustufen. Dieser Versuch sollte also nicht zur Bewertung der Heuristik herangezogen werden.

7.4 Zusammenfassung

In diesem Kapitel haben wir die Ergebnisse unserer Analyse mit Profildaten verglichen, und die Leistungsfähigkeit der Optimierung untersucht.

Dabei hat sich wie erwartet gezeigt, dass sich die absoluten, durch die Analyse geschätzten Häufigkeiten von den Profildaten unterscheiden. Ordnet man Felder bzw. Typen jedoch nach der Häufigkeit, ergeben sich durchaus ähnliche Sortierungen. Besonders pufferkritische Felder bzw. Klassen befinden sich in beiden Fällen oft oberhalb der Schwellenwerte. Unterschiede bei der Festlegung der Schwellenwerte ergeben sich durch die unterschiedlichen Abstufungen der Werte. Dies kompensiert die Analyse in einigen Fällen durch den Wechsel zu einer anderen Heuristik.

Aufgrund Analyse- wie Profildaten wählt die Heuristik offensichtlich unkritische Klassen für die Optimierung aus. Die richtigen Entscheidungen überwiegen jedoch und stimmen häufig überein. Im allgemeinen ist die Optimierung mit Analysedaten etwas konservativer, d.h., es wird weniger optimiert.

Wir haben die Laufzeitverbesserungen für verschiedene Optimierungsvarianten gemessen. In allen Fällen gibt es deutliche Laufzeitverbesserungen, aber auch Verschlechterungen. In den beiden Fällen, in denen sich die Optimierung mit Profildaten von der mit Analysedaten unterscheidet, nimmt die Heuristik eine Klasse zu wenig in einen Haufen auf, einmal mit Profildaten und das andere mal mit Analysedaten. Bei etwa der Hälfte der Testprogramme ergibt sich ein zu vernachlässigender Effekt, da sie nicht optimierbar sind, oder von vorneherein eine gute Pufferperformanz haben. Insgesamt überwiegen die Laufzeitverbesserungen.

Verbundteilen kommt nicht bei alle Programmen zur Anwendung. Richtiges Verbundteilen beeinflusst die Optimierung deutlich. Haben die Verbundtypen mehr pufferunkritische Felder als dies in den Testprogrammen der Fall ist, kann Verbundteilen Typanhäufen noch deutlicher zu Laufzeitverbesserungen verhelfen.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

In diese Arbeit wurde eine Optimierung vorgestellt, die die Pufferleistung von Zeigeranwendungen optimiert. Die grundlegenden Beiträge dieser Arbeit sind eine statische Analyse der Pufferprobleme und die Betrachtung von Vererbungshierarchien bei der Optimierung.

Die statische Analyse operiert auf der Zwischensprache des Übersetzers. Die für diese Arbeit weiterentwickelte Zwischensprache bietet eine Kombination aus wichtigen Informationen aus der Hochsprache, eine der Zielmaschine nahe Programmdarstellung und effizienten Datenstrukturen für die Analyse. So können Analyse und Pufferoptimierung von der Reduktion der Programmkomplexität durch andere Optimierungen wie Elimination toten und unerreichbaren Codes und Auflösen polymorpher Aufrufe profitieren. Andererseits stellt die Zwischensprache wichtige Details aus der Typisierung des Quellprogramms in der Zwischensprache dar. Dadurch können Datenstrukturen leicht analysiert, erkannt und verändert werden.

Diese Arbeit stellt zwei Optimierungen vor. Typanhäufen ist ein grobgranulares Verfahren um die Fehlzugriffsrate von Zeigeranwendungen zu reduzieren. Unsere Optimierung selektiert Typen für Typanhäufen, indem aus pufferkritischen Typen und pufferkritischen Kantenfeldern ein Haufengraph aufgebaut wird. Typen aus starken Zusammenhangskomponenten in diesem Graph werden in Typhaufen alloziert.

Verbundteilen erhöht die Effizienz von Typanhäufen. Durch Verbundteilen wird die räumliche Wiederverwendung in Typhaufen erhöht. Wenn diese

Wiederverwendung im Puffer als Lokalität ausgenutzt wird, reduzieren sich die Fehlzugriffe. Eine Heuristik betrachtet die pufferkritischen Felder eines Verbundes und entscheidet, ob ein Verbundtyp geteilt werden soll. Insbesondere berücksichtigt die Heuristik, dass weitere Typen in einer Vererbungshierarchie angepasst werden müssen, wenn ein Typ geteilt wird. Wir stellen verschiedene Verfahren vor, Typen zu teilen. Am erfolgreichsten ist optimales Verbundteilen, das Typinkompatibilitäten mit anderen Typen hinnimmt, wenn hier selten Typumwandlungen geschehen. Optimales Verbundteilen führt dann explizite Typumwandlungen ein. Beim Aufruf geerbter Methoden von durch die Optimierung inkompatibel gewordenen Obertypen werden Umschlagsmethoden generiert, die die Typanpassung vornehmen.

Beide Optimierungen setzen Informationen über pufferkritische Typen und Felder voraus. Diese werden mit einer statischen Analyse gewonnen. Diese Analyse schätzt die Größe der Datenstrukturen und die Häufigkeit der Zugriffe auf sie durch eine Analyse der Ausführungshäufigkeiten auf der Zwischendarstellung ab. Dazu werden Ausführungshäufigkeiten auf dem Steuerfluss der Methoden und auf dem Aufrufgraphen berechnet. Diese Analyse profitiert besonders von vorangegangenen Vereinfachungen des inter- und intraprozeduralen Steuerflusses. Um Datenstrukturen zu erkennen, auf denen häufige Iterationen stattfinden können, identifiziert die Analyse rekursive Kantenfelder in den Verbundtypen des Programms. Heuristiken legen Schwellenwerte für Typanhäufen und Verbundteilen fest. Dazu verwenden sie Allokationshäufigkeiten, Zugriffshäufigkeiten auf ganze Verbunde und Zugriffshäufigkeiten auf einzelne Felder. Dabei wird die Summe der Zugriffshäufigkeiten für einen Typen, sowie der Quotient mit der Allokationshäufigkeit betrachtet.

Der Vergleich unserer Heuristik mit Profildaten zeigt, dass sich aus den heuristischen Daten fast die gleichen Optimierungsentscheidungen ableiten lassen wie aus den Profildaten. Da Profildaten immer eine Ungenauigkeit durch die exemplarischen Eingaben des Messlaufs enthalten, ist die Heuristik den Profildaten ebenbürtig.

8.2 Bewertung

Wir konnten erstmals zeigen, dass die Pufferleistung dynamischer Datenstrukturen durch eine Übersetzeroptimierung in einem statischen Übersetzer optimiert werden kann. Dadurch werden rückkoppelnde Verfahren und der

durch sie entstehende Zusatzaufwand überflüssig. So muss für das zu optimierende Programm weder eine exemplarische Eingabe festgelegt werden, noch muss es ausgeführt werden. Durch unsere Optimierung wird es erstmals möglich auch bei der Übersetzung von Programmteilen Pufferoptimierungen dynamischer Datenstrukturen durchzuführen.

Die Optimierung kann bis zu 48% der Fehlzugriffe reduzieren, wodurch sich bis zu 31% Laufzeitgewinn ergeben. Für Anhäufen mit optimalem Verbundteilen ergibt sich im Schnitt ein Laufzeitgewinn von 2%. In der verwendeten Testprogrammmenge *jolden* sind einige, die nicht von Speicherzugriffen dominiert werden, also systematisch nicht von unserer Optimierung profitieren können. Die Heuristik zur Identifikation pufferkritischer Datenstrukturen stimmt gut mit den Messungen überein. So identifiziert sie bei 3 der 4 Testprogramme, die Pufferprobleme haben, die kritischen Datenstrukturen richtig. Der Optimierungseffekt wird nicht konsistent erreicht. Die Optimierung würde daher in einem Übersetzer sinnvoller Weise als zusätzliche Option implementiert.

Zusätzlich haben wir im Rahmen dieser Arbeit eine Zwischendarstellung entworfen, die Typsysteme von Hochsprachen mit Vererbung so darstellen, dass Veränderungen der Datentypen durch eine Optimierung leicht möglich sind. Dabei wird auf explizite Darstellung von Details des Typsystems Wert gelegt, die für die korrekte Übersetzung notwendig sind, und mit deren Wissen Analysen besser arbeiten können. Diese Darstellung kam bereits anderen Arbeiten zu Gute.

8.3 Ausblick

Obwohl die gewählten Testprogramme eine realistische Arbeitslast für eine Pufferoptimierung darstellen, wurde die Optimierung nicht auf echte Anwendungen angewandt. Eine weitere Untersuchung sollte sich mit großen Systemen auseinander setzen. Dazu sollte man Messmethoden entwickeln, die Performanzverbesserungen in einzelnen Programmteilen messen können, da Gewinne in einzelnen daten- und rechenintensiven Algorithmen nicht unbedingt auf die Programmlaufzeit durchschlagen. Diese kann zum Beispiel von externen Faktoren wie I/O dominiert sein. Durch eine Verbesserung einzelner Algorithmen kann jedoch trotzdem die Arbeitslast auf einem Server verbessert werden.

Die zum Vergleich herangezogene Optimierung mit Profildaten zeigt nicht immer einen positiven Effekt. Dies spricht dafür, die Heuristik für die Optimierungsentscheidungen weiter zu verbessern. Mit Profildaten kann man theoretisch die optimale Datenanordnung für ein Programm bei immer gleicher Eingabe berechnen. Da unsere Optimierung mit Hinblick auf heuristische Daten entworfen ist, kommt kein derartiges Verfahren in Betracht. Trotzdem sollte die Heuristik hier zu besseren Ergebnissen führen.

Durch die systematische Darstellung von Vererbung in unserer Zwischensprache können leicht weitere Optimierungen der Vererbungshierarchie formuliert werden. So kann an der Darstellung eines Programms in unserer Zwischensprache leicht abgelesen werden, ob ein Datentyp jemals zu seinem Obertyp umgewandelt wird. Ist dies nie der Fall, kann nach dem Einerben aller Felder und Methoden des Obertyps die Vererbungsrelationen zwischen dem Typ und seinem Obertyp entfernt werden. Dadurch ist in der Zwischensprache explizit dargestellt, das die zwei Typen im Programm nie ineinander Umgewandelt werden. Dies vereinfacht zum Beispiel Haldenanalysen, kommt aber auch Verbundteilen entgegen. Weitere Arbeiten sollten derartige Optimierungen und deren Effekt auf Verbundteilen und Typanhäufen untersuchen.

Die Literatur kennt verschiedene Verfahren, die auf Informationen über pufferkritische Verbunde und Felder aufbauen, zum Beispiel Optimierungen von Speicherbereinigern. Diese entnehmen ihre Information meist Laufzeitmessungen. Mit der in dieser Arbeit vorgestellten Heuristik können solche Verfahren vollständig in einen Übersetzer integriert werden.

Literaturverzeichnis

[AK84] J. R. Allen und Ken Kennedy: *Automatic loop interchange.* In: *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Kanada, Juni 1984.

[Ban88] U. Banerjee: *An introduction to a formal theory of dependence analysis.* The Journal of Supercomputing, 2(2):133–149, Oktober 1988.

[BBGL05] Michael Beck, Boris Boesler, Rubino Geiss und Götz Lindenmaier: *Firm, an Intermediate Language for Compiler Research.* Interner Bericht 2005-8, Fakultät für Informatik, Universität Karlsruhe (TH), März 2005.

[BCJ+94] David F. Bacon, J. Chow, Dz-ching R. Ju, K. Muthukumar und V. Sarkar: *A Compiler Framework for Restructuring Data Declarations to Enhance Cache and TLB Effectiveness.* In: *Proceedings of CASCON '94, Centre for Advanced Studies Conference*, Seiten 270–282, Toronto, Kanada, November 1994.

[BJWE91] F. Bodin, W. Jalby, D. Windheiser und C. Eisenbeis: *A Quantitative Algorithm for Data Locality Optimzation.* In: *Code Generation — Concepts, Tools, Techniques*, Seiten 119–145. Springer-Verlag, 1991.

[Boe05] Boris J. H. Boesler: *Codeerzeugung mit Graphersetzung und Lösungsgraphen.* Doktorarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Februar 2005.

[BS96] David F. Bacon und Peter F. Sweeney: *Fast static analysis of C++ virtual function calls.* In: *1996 ACM conference on*

Object–Oriented Programming Systems, Languages, and Applications, Band 31, Seiten 324–341, San Jose, CA, USA, Oktober 1996. ACM Press.

[BSW+00] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty und R. A. Davey: *A Benchmark Suite for High Performance Java*. Concurrency: Practice and Experience, 12(6):375–388, 2000.

[Cah02] Brendon Cahoon: *Effective Compile-Time Analysis for Data Prefetching in Java*. Doktorarbeit, Dept. of Computer Science, University of Massachusetts, Amherst, September 2002.

[Car96] Steve Carr: *Combining Optimization for Cache and Instruction-Level Parallelism*. In: *The 1996 International Conference on Parallel Architectures and Compilation Techniques*, Boston, MA, USA, Oktober 1996.

[CDL99] Trishul M. Chilimbi, Bob Davidson und James R. Larus: *Cache-Conscious Structure Definition*. In: *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, USA, 1999.

[CHL99] Trishul M. Chilimbi, Mark D. Hill und James R. Larus: *Cache-Conscious Structure Layout*. In: *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, USA, Mai 1999.

[CL95a] Steve Carr und R. B. Lehoucq: *A Compiler-Blockable Algorithm for QR Decomposition*. In: *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, USA, Februar 1995.

[CL95b] M. Cierniak und W. Li: *Unifying Data and Control Transformations for Distributed Shared-Memory Machines*. In: *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, Seiten 205–217, San Diego, CA, USA, Juni 1995.

[CM95] S. Coleman und Kathryn S. McKinley: *Tile Size Selection Using Cache Organization and Data Layout*. In: *Proceedings of*

the SIGPLAN '95 Conference on Programming Language Design and Implementation, Seiten 279–290, La Jolla, CA, USA, Juni 1995.

[CM01] Brendon Cahoon und Kathryn S. McKinley: *Data Flow Analysis for Software Prefetching Linked Data Structures in Java*. In: *The 2001 International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spanien, September 2001.

[CMT94] Steve Carr, Kathryn S. McKinley und C. Tseng: *Compiler Optimizations for Improving Data Locality*. In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Seiten 252–262, San Jose, CA, USA, Oktober 1994.

[DCS96] Chen Ding, Steve Carr und Phil Sweany: *Modulo Scheduling with Cache Reuse Information*. Technischer Bericht 96-07, Dept. of Computer Science, Michigan Technological University, September 1996.

[DCS97] Chen Ding, Steve Carr und Phil Sweany: *Modulo Scheduling with Cache Reuse Information*. In: *Proceedings of Euro-Par'97*, Band 1300 der Reihe *Lecture Notes in Computer Science*, Seiten 1079–1083, Passau, August 1997.

[Ess93] K. Esseghir: *Improving Data Locality for Caches*. Master Thesis, Dept. of Computer Science, Rice University, September 1993.

[FK98] Michael Franz und Thomas Kistler: *Splitting Data Objects to Increase Cache Utilization*. Technischer Bericht 98-34, Dept. of Information and Computer Science, University of California, Irvine, Oktober 1998.

[GJG88] D. Gannon, W. Jalby und K. Gallivan: *Strategies for Cache and Local Memory Management by Global Program Transformation*. Journal of Parallel and Distributed Computing, 5(5):587–616, Oktober 1988.

[GL01] Rubino Geiss und Götz Lindenmaier: *Global Configuration of Cache Optimizations.* In: *Proceeding of JOSES: Java Optimization Strategies for Embedded Systems*, Genua, Italien, April 2001. Technischer Bericht 10/01, Fakultät für Informatik, Universität Karlsruhe.

[GOST93] G. R. Gao, R. Olsen, V. Sarkar und R. Thekkath: *Collective Loop Fusion for Array Contraction.* In: U. Banerjee, D. Gelernter, A. Nicolau und D. Padua (Herausgeber): *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, Band 757 der Reihe *Lecture Notes in Computer Science*, Seiten 281–295. Springer-Verlag, 1993.

[GTZ98] Daniela Genius, Martin Trapp und Wolf Zimmermann: *An Approach to improve Locality using Sandwich Types.* In: *Proceedings of the 2nd Types in Compilation Workshop*, Seiten 194–214, Kyoto, Japan, März 1998.

[HRCR95] Laurie J. Hendren, John H. Reppy, Martin C. Carlisle und Anne Rogers: *Supporting Dynamic Data Structures on Distributed Memory Machines.* ACM Transactions on Programming Languages and Systems, 7(2), März 1995.

[jik] *The Jikes Project.* Internetseite, IBM, http://jikes.sourceforge.net/ faq/user-index.shtml.

[KE93] D. R. Kerns und S. Eggers: *Balanced Scheduling: Instruction Scheduling when Memory Latency is Uncertain.* In: *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Seiten 278–289, Albuquerque, NM, USA, Juni 1993.

[KF00] Thomas Kistler und Michael Franz: *Automated Data-Member Layout of Heap Objects to Improve Memory-Hierarchy Performance.* ACM Transactions on Programming Languages and Systems, 22(3):490–505, Mai 2000.

[KM93] Ken Kennedy und Kathryn S. McKinley: *Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and*

Distribution. In: U. Banerjee, D. Gelernter, A. Nicolau und D. Padua (Herausgeber): *Languages and Compilers for Parallel Computing*, Band 768 der Reihe *Lecture Notes in Computer Science*, Seiten 301–321, Portland, OR, USA, August 1993. Springer-Verlag.

[KP93] Wayne Kelly und William Pugh: *A Framework for Unifying Reordering Transformations*. Technischer Bericht UMIACS-TR-93-134, Dept. of Computer Science, Univ. of Maryland, College Park, April 1993.

[LA05] Chris Lattner und Vikram Adve: *Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap*. In: *Proceedings of the SIGPLAN '05 Conference on Programming Language Design and Implementation*, Chicago, IL, USA, Juni 2005.

[Lat05] Chris Lattner: *Macroscopic Data Structure Analysis and Optimization*. Doktorarbeit, Computer Science Dept., University of Illinois at Urbana-Champaign, Mai 2005.

[LDMM02] Teny Lyon, Eric Delano, Cameron McNairy und Dean Mulla: *Data Cache Design Considerations for the Itanium® 2 Processor*. In: *2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, September 2002.

[LE95] Jack L. Lo und Susan J. Eggers: *Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism*. In: *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, Seiten 151–162, San Diego, CA, USA, Juni 1995.

[LFL99] Richard Ladner, James D. Fix und Anthony LaMarca: *Cache Performance Analysis of Traversals and Random Accesses*. Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, Januar 1999.

[Li94] Wei Li: *Compiler Optimizations for Cache Locality and Coherence*. Technischer Bericht TR504, Computer Science Dept., University of Rochester, April 1994.

[Lin02] Götz Lindenmaier: *libFIRM – A Library for Compiler Optimization Research Implementing FIRM.* Interner Bericht 2002-5, Fakultät für Informatik, Universität Karlsruhe (TH), September 2002.

[Lin05] Götz Lindenmaier: *Structure Splitting and Inheritance.* Interner Bericht 2005-7, Fakultät für Informatik, Universität Karlsruhe (TH), März 2005.

[LL96] Anthony LaMarca und Richard E. Ladner: *The Influence of Caches on the Performance of Heaps.* Technischer Bericht TR-UW-CSE-96-02-03, Dept. of Computer Science, University of Washington, Seattle, Februar 1996.

[LL97] Anthony LaMarca und Richard E. Ladner: *The Influence of Caches on the Performance of Sorting.* Proceedings of the Eight Annual ACM-SIAM Symposium on Discrete Algorithms, Januar 1997.

[LM96] Chi-Kueng Luk und Todd C. Mowry: *Compiler-Based Prefetching for Recursive Data Structures.* In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Seiten 222–233, Boston, MA, USA, Oktober 1996.

[LM99a] Chi-Kueng Luk und Todd C. Mowry: *Automatic Compiler-Inserted Prefetching for Pointer-Based Applications.* IEEE Transactions on Computers, 48(2), 1999.

[LM99b] Chi-Kueng Luk und Todd C. Mowry: *Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation.* In: *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, USA, Mai 1999.

[LMT00] Götz Lindenmaier, Kathryn S. McKinley und Olivier Temam: *Load Scheduling with Profile Information.* In: Arndt Bode, Thomas Ludwig und Roland Wismüller (Herausgeber): *Proceedings of EuroPar'00*, Band 1900 der Reihe *Lecture*

Notes in Computer Science, Seiten 223–233, München, August 2000. Springer-Verlag.

[LRW91] M. S. Lam, E. Rothberg und M. E. Wolf: *The Cache Performance and Optimizations of Blocked Algorithms.* In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Seiten 63–74, Santa Clara, CA, USA, April 1991.

[LSKR95] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel und Robert R. Roediger: *SPAID: Software Prefetching in Pointer- and Call-Intensive Environments.* In: *Proceedings of the 28th International Symposium on Microarchitecture*, Ann Arbor, MI, USA, November 1995.

[LW92] Monica S. Lam und Robert P. Wilson: *Limits of Control Flow Parallelism.* In: *Proceedings of the 19th International Symposium on Computer Architecture*, Seiten 46–57, Gold Coast, Australia, Mai 1992.

[LW94] Alvin R. Lebeck und David A. Wood: *Cache Profiling and the SPEC Benchmarks: A Case Study.* IEEE Computer, 27(10):15–26, Oktober 1994.

[MT96] Kathryn S. McKinley und Olivier Temam: *A Quantitative Analysis of Loop Nest Locality.* In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Seiten 94–104, Boston, MA, USA, Oktober 1996.

[MT99] Kathryn S. McKinley und Olivier Temam: *Quantifying Loop Nest Locality Using SPEC'95 and the Perfect Benchmarks.* IEEE Transactions on Computer Systems, 17(4):288–336, Mai 1999.

[Muc97] Steven S. Muchnik: *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, 1997.

[PNDN97] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt und A. Nicolau: *Improving Cache Performance Through*

Tiling and Data Alignment. In: *Workshop on Parallel Algorithms for Irregularly Structured Problems*, Band 1253 der Reihe *Lecture Notes in Computer Science*, Seiten 167–185. Springer-Verlag, Paderborn, 1997.

[Rie04] Till Riedel: *Ausnahmebehandlung in einem Optimierenden Javaübersetzer*. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Juni 2004.

[RT98] Gabriel Rivera und Chau Wen Tseng: *Data Transformations for Eliminating Conflict Misses*. In: *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Kanada, Juni 1998.

[SAG+01] Artour Stoutchinin, José Nelson Amaral, Guang R. Gao, James C. Dehnert, Suneel Jain und Alban Douillet: *Speculative Prefetching of Induction Pointers*. In: *Proceedings of the 13th International Conference on Compiler Construction (???)*, Band 2027 der Reihe *Lecture Notes in Computer Science*, Seiten 289–303, Genua, Italien, April 2001. Springer-Verlag.

[SG97] F. Jesus Sanchez und Antonio Gonzalez: *Cache Sensitive Modulo Scheduling*. In: *The 1997 International Conference on Parallel Architectures and Compilation Techniques*, Seiten 261–271, San Francisco, CA, USA, November 1997.

[SGBS02] Y. Shuf, M. Gupta, R. Bordawekar und J. P. Singh: *Exploiting Prolific Types for Memory Management and Optimizations*. In: *Proceedings of the Twenty-nineth Annual ACM Symposium on the Principles of Programming Languages*, Band 37 der Reihe *ACM SIGPLAN Notices*, Seiten 295–306, New York, NY, USA, Januar 2002. ACM Press.

[SGF+02] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel und Jaswinder Pal Singh: *Creating and preserving locality of java applications at allocation and garbage collection times*. In: *2002 ACM conference on Object–Oriented Programming Systems, Languages, and Applications*, Band 37 der Reihe

ACM SIGPLAN Notices, Seiten 13–25. ACM Press, November 2002.

[Sin02] Sharad Singhai: *Data Reorganization for Improving Cache Performance of Object-Oriented Programs.* Doktorarbeit, Dept. of Computer Science, University of Massachusetts, Amherst, Februar 02.

[SM98] Sharad Singhai und Kathryn S. McKinley: *A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality.* The Computer Journal, 40(6):340–355, 1998.

[SN03] Julian Seward und Nick Nethercote: *Valgrind, version 2.0.0.* Technischer Bericht, http://developer.kde.org/~sewardj/, 2003.

[ST92] V. Sarkar und R. Thekkath: *A General Framework for Iteration-Reordering Loop Transformations (Technical Summary).* In: *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, Seiten 175–187, San Francisco, CA, USA, Juni 1992.

[SZ97] Matthew L. Seidl und Benjamin G. Zorn: *Predicting References to Dynamically Allocated Objects.* Technischer Bericht CU-CS-826-97, Dept. of Computer Science, University of Colorado at Boulder, Januar 1997.

[TGJ93] Olivier Temam, E. Granston und W. Jalby: *To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts.* In: *Proceedings of Supercomputing '93*, Seiten 410–419, Portland, OR, USA, November 1993.

[TLB99] Martin Trapp, Götz Lindenmaier und Boris Boesler: *Documentation of the Intermediate Representation FIRM.* Interner Bericht 1999-14, Fakultät für Informatik, Universität Karlsruhe (TH), Dezember 1999.

[Tra01] Martin Trapp: *Optimierung Objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.* Dok-

torarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Oktober 2001.

[WL91] Michael E. Wolf und Monica Lam: *A Data Locality Optimizing Algorithm.* In: *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Band 26(6), Seiten 30–44, Toronto, Kanada, Juni 1991.

[WLM92] Paul R. Wilson, Michael S. Lam und Thomas G. Moher: *Caching Consideration For Generational Garbage Collection.* In: *Conference on Lisp and Functional programming*, San Francisco, CA, USA, Juni 1992.

[Wol87] Michael J. Wolfe: *Iteration Space Tiling for Memory Hierarchies.* In: *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, Seiten 357–361, Los Angeles, California, Dezember 1987.

[Wol89] Michael J. Wolfe: *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and distributed Computing. MIT Press, Cambridge, MA, USA, 1989.

http://digbib.ubka.uni-karlsruhe.de/volltexte/1000007842