Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

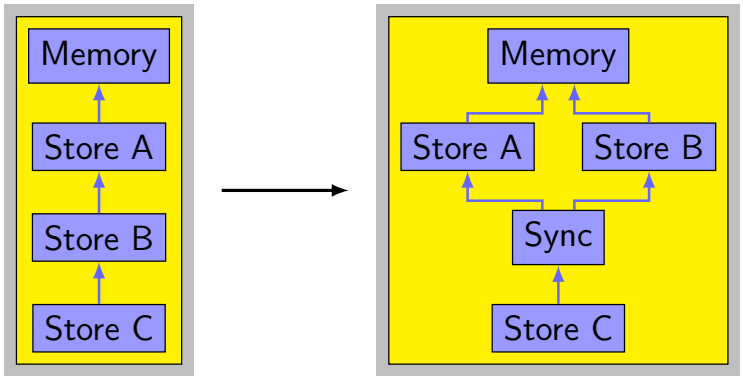
Lehrstuhl Programmierparadigmen
Prof. Dr.-Ing. Snelting

Repräsentation von Alias-Information in Programmgraphen

Diplomarbeit von

Christopher Frieler

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	-
Betreuender Mitarbeiter:	Dipl.-Inform. Matthias Braun

Bearbeitungszeit: 3. September 2012 – 18. Dezember 2012

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 18. Dezember 2012

Zusammenfassung

Moderne Compiler führen meistens auch eine Reihe von Optimierungen auf dem zu übersetzenden Programm durch. Dabei verwenden sie eine Zwischensprache für die Darstellung des Programms während der Optimierungen. Grundlage für viele dieser Optimierungen ist die Alias-Information, mit der sich Abhängigkeiten zwischen Speicherzugriffen erkennen oder ausschließen lassen. In dieser Arbeit wird die graphbasierte Zwischensprache Firm so erweitert, dass sie eine Repräsentation der Alias-Information unmittelbar in den Programmgraphen erlaubt. Darüber hinaus wird eine neue Optimierung entwickelt, die die Alias-Information in die Firm-Graphen einbringt. Dadurch benutzen nachfolgende Optimierungen automatisch auch diese Alias-Information, ohne sich explizit damit auseinandersetzen zu müssen.

Inhaltsverzeichnis

1 Einleitung **9**

2 Grundlagen **11**
2.1 Die Zwischensprache Firm 11
2.1.1 Static Single Assignment Form (SSA) 12
2.1.2 Speicher in Firm 14
2.2 Aliasing 16
2.2.1 Points-To-Analyse 16
2.2.2 Alias-Analyse in libFirm 17

3 Modellierung von Speicherzugriffen **19**
3.1 Parallelisierung mittels Alias-Information 19
3.2 Parallelisierung von Lesezugriffen 21
3.3 Modellierung von Funktionsaufrufen 23
3.4 Vorteile der neuen Modellierung 25

4 Verwandte Arbeiten **27**

5 Parallelisierungsverfahren **29**
5.1 Phase I: Speicheroperationen sammeln 29
5.2 Phase II: Speicherpartitionierung 30
5.2.1 Statische Partitionierung 31
5.2.2 Flusssensitive Partitionierung 32
5.2.3 Zusammenhang zwischen den Partitionierungsvarianten 37
5.3 Phase III: Bestimmen der neuen Abhängigkeiten 38
5.4 Einordnung in den Compiliervorgang 41
5.5 Abhängigkeitskanten in anderen Optimierungen 42

6 Optimierungen **45**
6.1 Laden eines gerade geschriebenen Wertes 45
6.2 Überschreiben eines gerade geschriebenen Wertes 46
6.3 Neuladen eines bereits geladenen Wertes 47
6.4 Eliminierung partieller Redundanzen 48
6.5 Schleifeninvariantes Laden eines Wertes 49
6.6 Überschreiben eines Wertes in Schleifen 50

7 Evaluation **53**
7.1 Programmlaufzeiten 53
7.2 Compilezeiten 56

8 Fazit **61**
8.1 Ausblick 62

Literatur **63**

1 Einleitung

Compiler übersetzen ein Programm, das in einer höheren Programmiersprache geschrieben ist, in Maschinencode für eine konkrete Rechnerarchitektur. Dabei unterscheiden sich die Zielsetzungen der Quell- und der Zielsprache voneinander. Höhere Programmiersprachen stellen ein Werkzeug für den Programmierer dar und sollen deshalb eine klare Strukturierung und die Verständlichkeit des Programms unterstützen, sowie die Möglichkeit bieten Berechnungen auf einem hohen Abstraktionsniveau kompakt auszudrücken. Für den Maschinencode ist hingegen in erster Linie die Performanz bei der Ausführung, also Laufzeit und Speicherbedarf wichtig. Zusätzlich zur reinen Übersetzung führen die meisten Compiler daher eine Reihe von Optimierungen bezüglich der Performanz-Ziele durch.

Die Optimierungen geschehen für gewöhnlich nicht direkt auf dem Quell- oder Zielprogramm. Stattdessen wird das Quellprogramm zunächst in eine Zwischensprache übersetzt, die durch spezielle Eigenschaften wie zum Beispiel durch SSA-Form die Optimierungen begünstigt. Erst nach den Optimierungen auf dieser Zwischendarstellung wird daraus der endgültige Maschinencode erzeugt. Eine Möglichkeit für die Zwischensprache ist die Darstellung des Programms durch Graphen. Eine solche graphbasierte Zwischensprache ist auch die in dieser Arbeit verwendete Sprache Firm.

Um das Programm in der Zwischensprache optimieren zu können muss der Compiler Informationen über das Programm gewinnen, indem er es analysiert. Eine wichtige Grundlage für viele Optimierungen ist dabei die Alias-Information. Sie gibt an, ob zwei Speicherzugriffe auf denselben Speicher zugreifen können, also aliasen, oder nicht. Damit lassen sich Abhängigkeiten zwischen diesen Speicherzugriffen erkennen oder ausschließen. Optimierungen, die von der Alias-Information profitieren sollen, mussten diese Information bisher fast immer explizit berücksichtigen.

In dieser Arbeit wird ein neues Verfahren vorgestellt, das die Alias-Information unmittelbar in der Struktur der Programmgraphen repräsentiert. Dadurch wird die Alias-Information automatisch von nachfolgenden Optimierungen genutzt, ohne dass sie dafür explizit betrachtet werden muss. Zu diesem Zweck wird eine neue Optimierung entwickelt, die auf Basis der Alias-Information den vom Programm benutzten Speicher aufteilt, sodass unabhängige Speicherzugriffe unterschiedliche Speicherbereiche benutzen. Dabei wird auch die Zwischensprache Firm so modifiziert, dass sie die Modellierung einzelner Speicherbereiche unterstützt. Gleichzeitig wird die Modellierung von Lesezugriffen verbessert.

Kapitel 2 stellt zunächst die Grundlagen zur Zwischensprache Firm und zum Aliasing vor. In Kapitel 3 wird dann die neue Modellierung des Speichers in Firm erarbeitet. Diese neue Modellierung stellt gleichzeitig das Zielformat für das später entwickelte Verfahren zur Repräsentation der Alias-Information dar. In Kapitel 4 sind einige Arbeiten, die sich bereits mit diesem Themengebiet beschäftigen, zusammengefasst. Allerdings gibt es in diesem Bereich bisher wenig Forschungsarbeiten. In Kapitel 5 wird dann ein Verfahren entwickelt, mit dem sich der Speicher aufteilen lässt und die Alias-Information in Firm-Graphen eingebracht wird. Kapitel 6 betrachtet die Vorteile der Neuerungen für andere Optimierungen. Die Auswirkungen auf die Performanz der erzeugten Programme und des Compilers werden in Kapitel 7 untersucht. Kapitel 8 schließlich fasst die Ergebnisse der Arbeit zusammen und gibt Anregungen für mögliche Fortführungen.

2 Grundlagen

In diesem Kapitel wird die graphbasierte Zwischensprache Firm kurz beschrieben. Das in Kapitel 5 vorgestellte Verfahren zur Repräsentation von Alias-Information arbeitet auf Programmgraphen in Firm. Für die Implementierung des Verfahrens wurde libFirm verwendet, eine Implementierung der Zwischensprache Firm in der Programmiersprache C. Im zweiten Abschnitt dieses Kapitels werden Aliasing und zwei verschiedene Formen der Alias-Analyse betrachtet.

2.1 Die Zwischensprache FIRM

Firm [LBBG05] ist eine graphbasierte Zwischensprache für Computerprogramme. Sie dient in erster Linie Compilern, um Quellprogramme zu analysieren, zu optmieren und in Maschinencode zu übersetzen. Firm basiert wesentlich auf den von Click und Paleczny beschriebenen Konzepten [CP95].

Die Repräsentation eines Programms in Firm besteht aus je einem Graph für jede Funktion des Programms. Diese Programmgraphen besitzen die Grundblöcke und die einzelnen Instruktionen der Funktion als Knoten. Jeder Instruktionsknoten ist einem Grundblock zugeordnet. Neben der eigentlichen Instruktion repräsentiert jeder Instruktionsknoten auch den Wert, den diese Instruktion liefert. Liefert eine Instruktion mehrere Werte, so werden diese als Tupel aufgefasst und die einzelnen Komponenten mit Projektionsknoten extrahiert. Jeder Knoten ist mit den Werten, die er benutzt, bzw. mit deren definierenden Instruktionsknoten über eine Kante verbunden. Da es sich bei Firm-Graphen stets um Abhängigkeitsgraphen handelt, sind diese Kanten gerichtet und verlaufen immer vom Benutzer eines Wertes zu dessen Definition. Trotz dieser Kantenrichtung sprechen wir von einem „Eingang“ für diesen Wert am benutzenden Knoten. In Firm wird auch der Steuerfluss als Wert aufgefasst. Daher hat jeder Grundblock die Knoten als Vorgänger, die den ihn erreichenden Steuerfluss definieren.

Abbildung 2.1 zeigt eine kleine C-Funktion, die zwei als Parameter übergebene Zahlen addiert, sowie den zugehörigen Firm-Graphen zu dieser Funktion. Die Grundblöcke sind gelb eingezeichnet und enthalten die eigentlichen Instruktionen. Zusätzlich zum mittleren Grundblock, der den Rumpf der Funktion bildet, gibt es noch einen Start- und einen Endblock. Der Startblock enthält den Start-Knoten, die Quelle des gesamten Daten- und Steuerflusses innerhalb der Funktion. Aus diesem Knoten werden mittels Projektions-

knoten der initiale Speicherzustand (Proj M), der initiale Kontrollfluss (Proj X) und die Argumente (Proj T), die selbst wieder ein Tupel aus zwei Werten sind, extrahiert. Der mittlere Block besitzt den initialen Kontrollfluss als Vorgänger, d.h., er wird sofort nach betreten der Funktion erreicht. In diesem Block befindet sich ein Add-Knoten, der von den beiden Argumenten abhängt und das Ergebnis der Addition dieser Werte repräsentiert. Danach folgt ein Return-Knoten, der das Ergebnis der Addition sowie den unveränderten Speicherzustand als Ergebnis der Funktion zurückgibt. Der Graph wird durch den Endblock mit einem End-Knoten abgeschlossen. Er bildet die Senke für jeglichen Steuerfluss in der Funktion.

Zusätzlich zu ihrem Namen besitzen viele Knoten noch einen „Mode“, der den Typ ihres Wertes beschreibt und in der graphischen Darstellung als Kürzel im Knoten vermerkt ist. Im Beispiel kommen der Mode M für den Speicherzustand, X für den Steuerfluss, T für Tupel und Is für vorzeichenbehaftete Integerwerte vor.

```
1  int add(int a, int b)
2  {
3      return a + b;
4  }
```

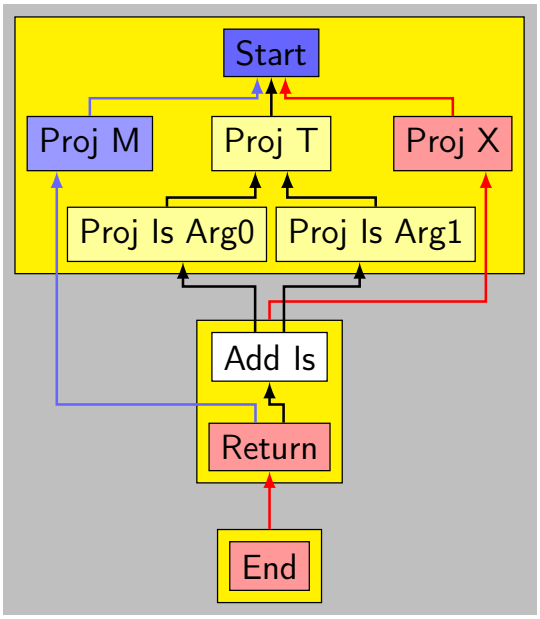


Abbildung 2.1: Eine C-Funktion zur Addition von zwei Zahlen und der zughörige Firm-Graph.

2.1.1 Static Single Assignment Form (SSA)

Eine wichtige Eigenschaft von Firm ist, dass alle Funktionsgraphen stets in SSA-Form sind. SSA steht für statische Einmalzuweisung (engl.: static single assignment). Das bedeutet, es gibt zu jeder Programmvariablen nur eine Zuweisung im Programmtext. Damit ist auch an jeder Verwendungsstelle unmittelbar klar, welcher Wert benutzt wird, nämlich

der, der an der einzigen Zuweisungstelle zugewiesen wurde. Das vereinfacht viele Analysen und Optimierungen. In FIRM gibt es keine Variablennamen mehr um auf Werte zuzugreifen. Stattdessen wird ein Wert in Form des Knotens, der ihn definiert, referenziert. Die SSA-Eigenschaft ist in FIRM dadurch gegeben, dass, wenn ein Knoten von einem Wert abhängt, er genau eine Kante zu einem anderen Knoten hat, der diesen Wert definiert. Dadurch ist auch hier der benutzte Wert sofort ersichtlich.

Enthalten zwei Programmpfade verschiedene Definitionen eines Wertes und laufen diese Pfade dann zusammen, so ist es nicht möglich, einer Verwendung nach dem Zusammenlaufen eine einzige Definition zuzuordnen. Abbildung 2.2 zeigt ein Beispiel für diese Situation. Am unteren Grundblock laufen zwei Programmpfade zusammen, die beide eine Definition für den Rückgabewert enthalten. Es lässt sich keine eindeutige Definitionsstelle festlegen. Als Lösung muss ein zusätzlicher Phi-Knoten eingefügt werden. Der Phi-Knoten wählt abhängig vom Vorgänger, über den sein Block erreicht wird, den Wert am ensprechenden Eingang aus. Im Beispiel liefert der Phi-Knoten also den Wert der Addition, wenn vorher der linke Block ausgeführt wurde, und sonst den Wert der Subtraktion.

```
1  if (...)
2      x = a + b;
3  else
4      x = a − b;
5
6  return x;
```

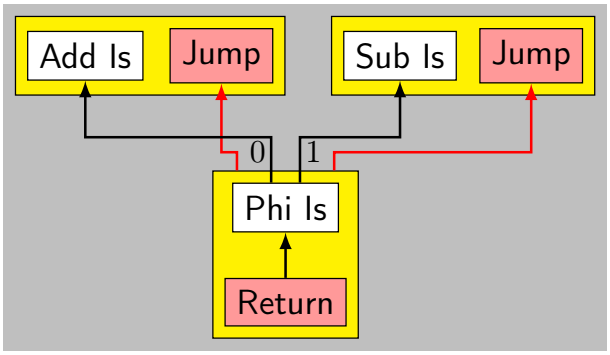


Abbildung 2.2: Nach dem if-Konstrukt laufen Prorgammpfade zusammen. Dadurch wird ein Phi-Knoten nötig, der über die gültige Definition von x entscheidet.

Zum Einfügen der Phi-Knoten gibt es verschiedene Strategien. In [CFR+91] wird ein Verfahren vorgestellt, bei dem nur die minimal nötige Anzahl an Phis eingefügt wird. Das Verfahren nutzt aus, das Phi-Knoten nur an den Dominanzgrenzen vorkommen können. Dazu muss allerdings der Programmgraph bereits soweit aufgebaut sein, dass die Dominanzgrenzen berechnet werden können. Die dieser Arbeit zugrunde liegende FIRM-Implementierung libFIRM verwendet daher ein Verfahren, bei dem an allen Stellen, an denen Programmpfade zusammenlaufen, vorläufige Phi-Knoten eingefügt werden. Diese können dann später wieder entfallen. Dadurch können bereits beim Graphaufbau Phi-Knoten erzeugt und die SSA-Form sichergestellt werden.

Eine weitere wichtige Eigenschaft der SSA-Form ist, dass der Grundblock jeder Verwendungsstelle eines Wertes immer vom Grundblock der Definitionsstelle dominiert wird. Damit ist sichergestellt, dass der verwendete Wert auch existiert. Wird dieses Dominanzkriterium verletzt, so liegt in den meisten Programmiersprachen ein Fehler im Quellpro-

gramm vor, etwa eine nicht initialisierte Variable. In Firm kann für solch einen fehlenden Wert ein Unknown-Knoten eingefügt werden.

2.1.2 Speicher in FIRM

Im Beispiel am Anfang dieses Kapitels wurde bereits der Speicherzustand erwähnt. Der Speicherzustand wird in Firm ebenfalls als Wert modelliert und deshalb im Folgenden synonym auch als Speicherwert bezeichnet. Abbildung 2.3 zeigt einen Firm-Graph mit Speicheroperationen. Der initiale Speicherwert steht am Anfang einer Funktion gemeinsam mit den Parametern im Start-Knoten zur Verfügung. Wenn die Funktion an einem Return-Knoten wieder verlassen wird, braucht dieser Return-Knoten neben einem möglichen Rückgabewert immer auch den letzten Speicherzustand.

Jeder Knoten, der innerhalb der Funktion auf den Speicher zugreift, benutzt zusätzlich zu seinen normalen Operanden den aktuellen Speicherzustand und definiert einen neuen. Load-Knoten lesen einen Wert aus dem Speicher. Sie brauchen eine Adresse und einen Speicherwert und liefern ein Tupel aus einem neuen Speicherwert und dem Wert, der im alten Speicherzustand an der Adresse stand. Analog schreiben Store-Knoten einen Wert in den Speicher. Sie brauchen eine Adresse, den zu schreibenden Wert und einen Speicherwert und definieren einen neuen Speicherwert. Da jede Funktion einen initialen Speicherwert braucht und einen finalen Speicherwert zurückgibt, haben auch Funktionsaufrufe einen Speicherein- und -ausgang. Funktionsaufrufe werden durch Call-Knoten modelliert.

Die Bedeutung des Speicherwertes geht in Firm allerdings noch über den reinen Zustand des Hauptspeichers hinaus. Der Speicherwert wird grundsätzlich benutzt um Instruktionen, deren Effekte auch außerhalb des Programms beobachtbar sind, in eine feste Reihenfolge zu bringen. Ebenso wird durch den Speicherwert verhindert, dass diese Instruktionen durch Optimierung wegfallen, weil es immer einen Benutzer des von ihnen definierten Speicherwertes gibt, diese Instruktionen also benötigt werden. Zu den beobachtbaren Effekten gehören neben dem Zustand des Speichers auch Ein- und Ausgabeoperationen sowie die Termination des Programms. Das ist ein weiterer Grund, weswegen Call-Knoten einen neuen Speicherwert definieren müssen, denn sie könnten Ein- und Ausgaben machen oder eine Endlosschleife enthalten. Der Speicherwert kann auch im Zusammenhang mit Exceptions benutzt werden. So besitzen auch Div- und Mod-Knoten – sie modellieren Division und Modulo-Operation – einen Speicherausgang, weil die Division durch Null eine Exception hervorruft, die von außen beobachtet werden kann. Die mögliche Exception kann aber auch explizit im Steuerfluss modelliert werden oder sie wird gänzlich ignoriert, wenn eine Division durch Null gemäß Spezifikation der Quellsprache nur zu undefiniertem Verhalten führt.

Genau wie alle anderen Werte in Firm erfüllt auch der Speicherwert die SSA-Eigenschaft. Das bedeutet, auch für den Speicherwert sind in Firm Phi-Knoten nötig, wenn

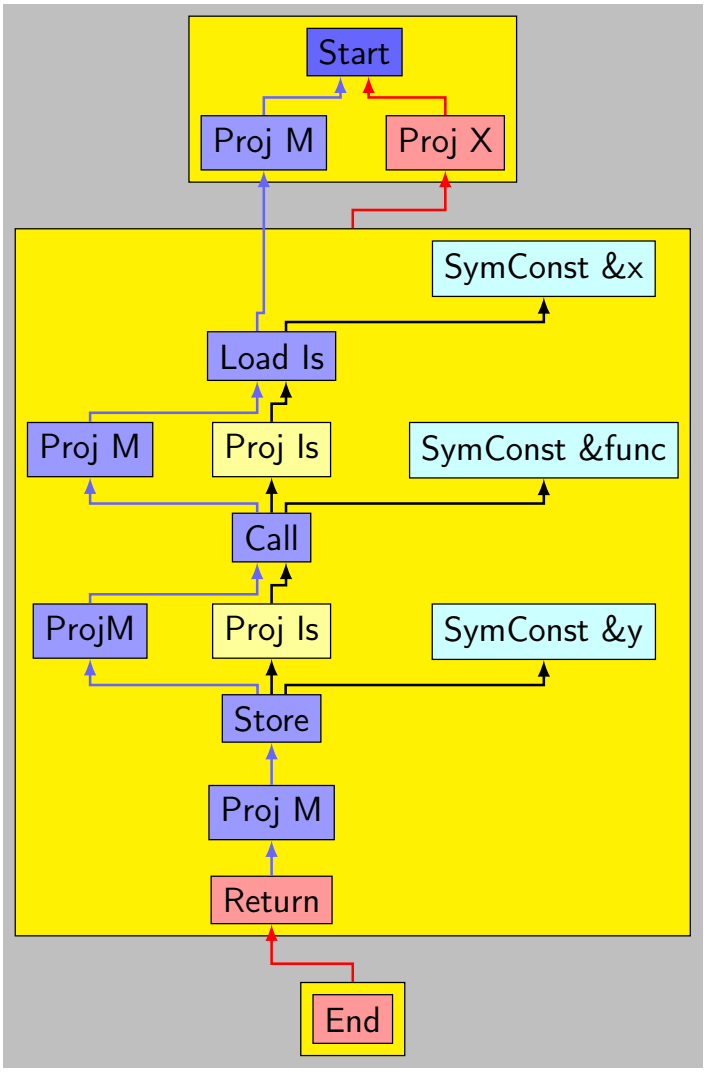


Abbildung 2.3: Ein FIRM-Graph mit Speicheroperationen. Jede Speicheroperation benötigt einen Speicherwert und definiert einen neuen. Speicher wird in FIRM-Graphen traditionell blau dargestellt.

mehrere Programmpfade zusammenlaufen, auf denen unterschiedliche Speicherwerte definiert werden, und die Definitionsstelle eines Speicherwertes dominiert stets alle seine Verwendungsstellen.

2.2 Aliasing

Viele Programmiersprachen verfügen über Zeiger oder Referenzen. In Java etwa sind alle Objektvariablen grundsätzlich Referenzen. Darüber hinaus ist es jedoch nicht möglich, selbst Referenzen auf beliebige Daten zu erzeugen. Die Programmiersprache C hingegen bietet dem Programmierer weitaus größere Freiheiten: Mit dem &-Operator kann die Adresse eines Speicherinhalts ermittelt und mit dem $*$-Operator wieder dereferenziert werden. C erlaubt sogar Zeigerarithmetik, also das Berechnen neuer Zeiger aus Zeigern.

Durch diese Zeigermechanismen entstehen verschiedene Zugriffsmöglichkeiten für eine Speicherstelle. Beispielsweise kann nach der C-Anweisung p = &x; sowohl mit x = ...; als auch mit $*$p = ...; der Wert von x geändert werden. Dieses Phänomen wird als Aliasing bezeichnet und wir sagen, zwei Speicherzugriffe „aliasen“, wenn sie auf den gleichen Speicher zugreifen können. Zusätzlich kann auch ein einzelner Zeiger abhängig vom konkreten Programmablauf auf verschiedene Speicherstellen zeigen. Insgesamt wird es dadurch schwierig zu erkennen, ob zwei Speicherzugriffe auf den gleichen Speicherbereich zugreifen oder ob sie unabhängig sind. Im Folgenden werden zwei Ansätze der Alias-Analyse vorgestellt.

2.2.1 Points-To-Analyse

Das Prinzip jeder Points-To-Analyse ist es, zu jedem Zeiger die Menge aller Datenobjekte, auf die er zeigen kann, auszurechnen. Diese Menge wird Points-To-Menge genannt. Als Datenobjekte werden alle statisch allokierten Daten sowie ein Repräsentant für jede Programmstelle mit dynamischer Speicherallokation benutzt. Zwei Speicherzugriffe sind genau dann unabhängig, wenn die Points-To-Mengen der verwendeten Zeiger disjunkt sind. Sind ihre Points-To-Mengen hingegen nicht disjunkt, aliasen die beiden Speicherzugriffe.

Da die Berechnung der Points-To-Mengen sehr aufwändig ist – in touringmächtigen Sprachen sind die exakten Points-To-Mengen sogar unentscheidbar [Lan92] –, gilt hier das Prinzip der konservativen Approximation: Die berechneten Mengen dürfen zum exakten Ergebnis noch zusätzliche Elemente enthalten. Zu große Abweichungen sind aber selbstverständlich unerwünscht, da so zusätzliche Abhängigkeiten zwischen den Speicherzugriffen berücksichtigt werden müssen. Die beiden wichtigsten Verfahren zur Points-To-Analyse stammen von Andersen [And94] und Steensgaard [Ste96].

2.2.2 Alias-Analyse in libFIRM

Die verwendete FIRM-Implementierung libFIRM bietet bereits eine Alias-Analyse. Diese verfolgt jedoch einen anderen Ansatz als klassische Points-To-Analysen und es werden keine Points-To-Mengen berechnet. Stattdessen werden bei der Abfrage, ob zwei Speicherzugriffe in Form ihrer FIRM-Knoten aliasen, einige Regeln zur Ableitung von Alias-Information geprüft.

- Sind die FIRM-Knoten, die die Adressen repräsentieren gleich, so aliasen die Speicherzugriffe auf jeden Fall.
- Sind die Adressknoten das Ergebnis von Zeigerarithmetik, so werden die Basisadressen und Offsets verglichen. Bei gleicher Basisadresse und konstanten Offsets kann so eine Überlappung der benutzten Speicherbereiche erkannt oder ausgeschlossen werden.
- Wird auf unterschiedliche Felder eines zusammengesetzten Typs (etwa ein struct in C) zugegriffen, so können die Zugriffe nicht aliasen. Ebenso können Felder, lokale und globale Variablen nicht miteinander aliasen und globale Variablen untereinander auch nicht.
- In typsicheren Sprachen können auch Zugriffe über Zeiger mit unterschiedlichen Typen nicht aliasen. [1]

Ist keine Regel anwendbar, so gilt auch hier das Prinzip der konservativen Approximation und das Ergebnis der Analyse ist, dass die beiden Speicherzugriffe aliasen können.

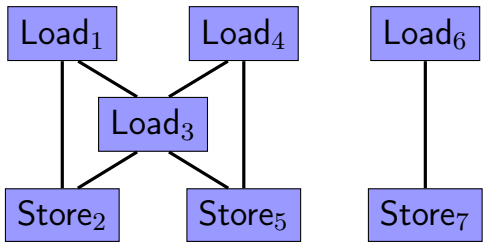


Abbildung 2.4: Ein Beispiel für einen Alias-Graph. Die zugrunde liegende Alias-Relation ist nicht transitiv.

Das Ergebnis der Alias-Analyse für alle Speicherzugriffe ist die Alias-Relation, eine binäre Relation auf der Menge der Speicherzugriffe. Sie lässt sich auch als Graph auffassen. Dieser Alias-Graph besitzt die Speicherzugriffe als Knoten. Zwei Knoten sind genau dann durch eine (ungerichtete) Kante verbunden, wenn sie aliasen können. Abbildung 2.4 zeigt

[1] Die Programmiersprache C erlaubt zwar beliebige Konversion von Zeigern (mit Ausnahme von Funktionszeigern), ein Zugriff über einen Zeiger, der durch Konversion zu einem inkompatiblen Typen entstanden ist, führt aber zu undefiniertem Verhalten.

ein Beispiel für einen Alias-Graph. Offensichtlich ist die Alias-Relation im Allgemeinen nicht transitiv. Wird etwa über einen Zeiger auf eine von zwei möglichen Variablen zugegriffen und gibt es auch direkte Zugriffe auf diese beiden Variablen, so aliasen zwar die beiden direkten Zugriffe mit dem über einen Zeiger, jedoch nicht untereinander und die Transitivität ist verletzt.

3 Modellierung von Speicherzugriffen

In Abschnitt 2.1.2 wurde die bisherige Modellierung von Speicher in Firm beschrieben. Jeder Speicherzugriff benötigt den Speicherwert des vorangegangenen Zugriffs und erzeugt einen neuen, der vom nächsten Speicherzugriff benutzt werden muss. Durch die Speicherkanten wird eine Totalordnung aller Speicherzugriffe auf einem Ausführungspfad definiert. Lediglich durch Aufteilung des Steuerflusses kann ein Speicherwert von den jeweils ersten Speicherzugriffen auf verschiedenen Ausführungspfaden gleichzeitig benutzt werden. Durch die Totalordnung entstehen unnötige Reihenfolge-Abhängigkeiten zwischen Speicherzugriffen, die nicht aliasen. In diesem Kapitel wird deshalb eine erweiterte Modellierung des Speicherzustandes entwickelt, die diese Totalordnungen zu einer Halbordnung abschwächt. Dazu wird die verfügbare Alias-Information genutzt, um unabhängige Speicherzugriffe zu erkennen, und gleichzeitig wird diese Information im Programmgraph repräsentiert.

3.1 Parallelisierung mittels Alias-Information

Kernidee der neuen Modellierung ist es, den Speicher in disjunkte Partitionen zu zerlegen. An die Stelle des einen Speicherwertes für den gesamten Speicher tritt dann ein Speicherwert für jede Partition des Speichers. Jeder Speicherzugriff benötigt nur noch die Speicherwerte genau der Partitionen, auf die er tatsächlich zugreifen kann. Auch werden nur für diese Partitionen neue Speicherwerte erzeugt. Der Start-Knoten definiert die initialen Speicherwerte für alle Partitionen. Ebenso benötigt ein Return-Knoten die finalen Speicherwerte für alle vorkommenden Partitionen.

Benötigt ein Speicherzugriff mehrere Speicherwerte, weil er entweder auf mehrere Partitionen zugreift oder die zugegriffene Partition nicht eindeutig ermittelt werden kann, und werden diese Speicherwerte von verschiedenen Firm-Knoten definiert, so werden die benötigten Speicherwerte erst in einem Sync-Knoten zusammengeführt. Auf diese Weise hat jeder Speicherzugriff weiterhin genau einen Speichereingang. Ein Sync-Knoten erzeugt aber selbstverständlich keinenerlei Instruktionen im Maschinencode. Da ein Speicherzugriff auch mehrere Speicherwerte definieren kann, können in der neuen Modellierung auch mehrere Firm-Knoten, die verschiedene dieser Speicherwerte benötigen, gleichzeitig von einem Speicherzugriff abhängen. So etwas war bisher nur möglich, wenn sich der Steuerfluss aufteilt und die abhängigen Knoten auf verschiedenen Ausführungspfaden liegen. Zur Unterscheidung könnte ein Split-Knoten eingeführt werden, der analog zu einem

Sync-Knoten einen Speicherwert aufteilt. Allerdings ergibt sich außer der genaueren Modellierung bisher kein Vorteil aus der Einführung von Split-Knoten. Stattdessen würde ein solcher Knoten die Programmgraphen vergrößern und komplizierter machen. Man muss sich jedoch stets vor Augen halten, dass die Situation eines Speicherzugriffes, von dessen Speicherwert mehrere Knoten abhängen, immer zwei mögliche Gründe haben kann: die Aufteilung des Steuerflusses oder die Aufteilung des Speichers. Es kann sogar beides gleichzeitig an einem Knoten auftreten.

Durch die erweiterte Modellierung können Speicherzugriffe, die nicht aliasen, in der Knoten-Ordnung, die durch die Speicherkanten gegeben ist, parallelisiert werden. Das reduziert die Abhängigkeiten, die zwischen den FIRM-Knoten bestehen. Abbildung 3.1 zeigt einen Ausschnitt aus einem FIRM-Graph nach einer solchen Parallelisierung. Die Knoten Store_1 und Store_2 aliasen nicht. Sie greifen auf unterschiedliche Speicherpartitionen zu und benutzen deshalb zwei unterschiedliche Speicherwerte, die jedoch beide von einem Knoten definiert wurden. Dadurch ist ihre Reihenfolge egal und sie liegen parallel; die Abhängigkeit zwischen ihnen besteht nicht mehr. Store_3 aliast sowohl mit Store_1 als auch mit Store_2 und benötigt daher die Speicherwerte, die von den beiden Knoten definiert werden. Sie werden vorher mit einem Sync-Knoten vereinigt.

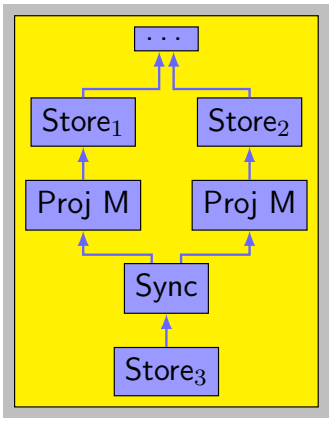


Abbildung 3.1: Ausschnitt aus einem FIRM-Graph nach der Speicherparallelisierung: Store_1 und Store_2 aliasen nicht, Store_3 aliast hingegen mit Store_1 und Store_2.

Für die Genauigkeit, mit der die verfügbare Alias-Information im Programmgraph repräsentiert wird, spielt die Granularität der Partitionierung eine entscheidende Rolle. Je feiner die Partitionierung ist, desto genauer lässt sich die Alias-Information darstellen. Allerdings nützt es nichts, feiner zu partitionieren als für die Darstellung der Alias-Information nötig. So bringt es etwa keinen Vorteil zwei Speicherpartitionen, die immer nur gemeinsam von Speicherzugriffen benötigt und neu definiert werden, getrennt zu halten. Im Gegenteil bläht eine zu feine Partitionierung den Graph unnötig mit zusätzlichen Kanten und Sync-Knoten auf. Schließlich braucht auch jeder Speicherwert seine

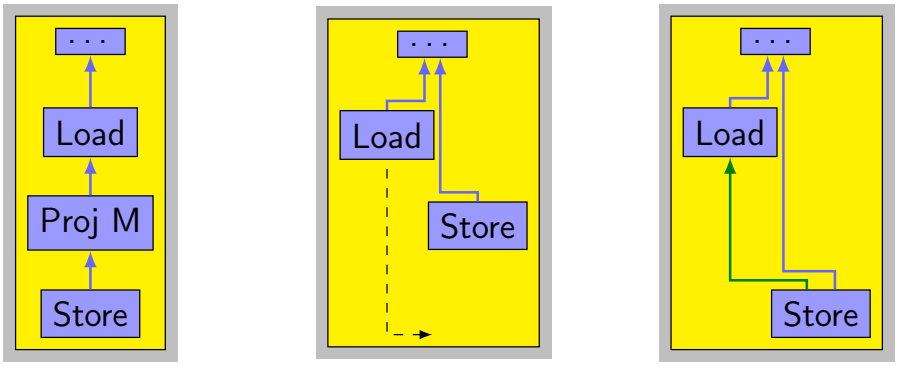
eigenen Phi-Knoten, was den Graph weiter vergrößert und das Herstellen der SSA-Form aufwändiger macht. Insgesamt gilt auch bei der Partitionierung des Speichers das Prinzip der konservativen Approximation. Zu einer gültigen Speicherpartitionierung ist auch jede Partitionierung erlaubt, in der mehrere Partitionen zu einer zusammengefasst sind. Durch eine gröbere Partitionierung werden die Darstellung der Alias-Information und darauf aufbauende Optimierungen zwar ungenauer, aber das Einarbeiten der Alias-Information in den Programmgraph wird einfacher und schneller. Das in Kapitel 5 vorgetellte Verfahren orientiert sich deshalb bei der Partitionierung des Speichers an den vorkommenden Speicherzugriffen und ihrer Alias-Relation, um nicht unnötig fein zu partitionieren.

3.2 Parallelisierung von Lesezugriffen

Eine weitere Modifikation der Modellierung von Speicher in Firm betrifft lesende Speicherzugriffe, also Load-Knoten. Bisher erzeugten auch diese einen neuen Speicherwert. Tatsächlich verändert ein lesender Zugriff den Speicherzustand aber nicht. Deshalb haben Load-Knoten in der neuen Modellierung keinen Speicherausgang mehr. Nachfolgende Speicheroperationen benutzen stattdessen den selben Speicherwert wie der Load-Knoten. Dadurch werden aufeinander folgende Lesezugriffe parallelisiert, auch wenn sie aliasen. Eine Ausnahme bilden Load-Knoten, die als „volatile“ gekennzeichnet wurden. Solche Lesezugriffe lesen Speicher, der auch von außerhalb des Programms oder von einem anderen Thread verändert werden kann. Da der gelesene Wert den Ablauf und das Ergebnis des Programms beeinflussen kann, stellt ein solcher Lesezugriff einen beobachtbaren Effekt dar, der nicht aufgrund von Optimierungen verschoben oder entfernt darf. Diese Load-Knoten müssen deshalb doch einen neuen Speicherwert definieren. Bei der Neumodellierung der Lesezugriffe ist noch zu beachten, dass dadurch eine dritte Möglichkeit entsteht, wie mehrere Instruktionen denselben Knoten als Speichereingang benutzen. Neben der Aufteilung des Steuerflusses und der Partitionierung des Speichers können nun auch noch Load-Knoten ein Duplikat eines von einem anderen Knoten verwendeten Speicherwertes benutzen.

Die Ordnung der Knoten wird durch das Entfernen der Speicherausgänge allerdings zu weit abgeschwächt. Ein Lesezugriff dürfte so hinter einen Schreibzugriff auf den gleichen Speicherbereich verschoben werden, weil zwischen diesen beiden Knoten keine Abhängigkeit in Form einer Kante existiert. Dann würde aber der Lesezugriff den manipulierten, also einen falschen Wert aus dem Speicher lesen. Abbildung 3.2 veranschaulicht diese Problematik. Die Modellierung wäre zwar insofern korrekt, als dass der Load-Knoten den richtigen Speicherzustand verwendet, dieser müsste aber während der Programmausführung als Kopie bis nach dem Schreibzugriff gesichert werden. Bei einem skalaren Wert wäre dies kein Problem: man würde eine Kopie in einem Register oder – falls die Register nicht ausreichen – im Hauptspeicher behalten. Der Speicherzustand passt jedoch weder in ein Register noch kann er im Hauptspeicher gesichert werden. Um diesem Problem zu begegnen wird eine zusätzliche Kante vom Schreibzugriff zum Lesezugriff eingefügt,

die die richtige Reihenfolge sicherstellt. Diese Kanten heißen Abhängigkeitskanten und werden in Firm-Graphen grün visualisiert.



(a) Klassiche Modellierung (b) Falsche Modellierung (c) Richtige Modellierung

Abbildung 3.2: Ein Load vor einem Store. In der klassischen Variante (a) besitzt das Load noch einen Speicherausgang. Um zu verhindern, dass das Load hinter das Store rutscht (b), wird eine Abhängigkeitskante eingefügt (c).

Abhängigkeitskanten modellieren lediglich Reihenfolge-Abhängigkeiten zwischen Firm-Knoten. Im Gegensatz zu anderen bisher betrachteten Abhängigkeiten liegt hier kein Wert (nicht einmal Speicher oder Kontrollfluss) zugrunde, der vom Zielknoten der Kante definiert und von ihrem Quellknoten gebraucht wird. Deshalb erfüllen Abhängigkeitskanten auch nicht die SSA-Eigenschaft. Es gibt also weder Phi-Knoten für diese Kanten noch muss der Quellknoten der Kante vom Zielknoten dominiert werden. Die fehlende SSA-Eigenschaft bietet einen enormen Vorteil, wenn ein Phi-Knoten eingefügt werden müsste, jedoch auf einem Ausführungspfad gar kein Lesezugriff erfolgt. Dann existiert für den zugehörigen Eingang des Phi-Knotens kein Zielknoten oder aber es müsste ein künstlicher Knoten eingefügt werden, der in irgendeiner Form die Abwesenheit jeglicher Abhängigkeitsziele symbolisiert.

Bei der Ermittlung eines geeigneten Quellknotens für eine Abhängigkeitskante kann nun vor einem Schreibzugriff ein Phi M-Knoten kommen, wenn dort verschiedene Ausführungspfade zusammenlaufen. Für einen normalen SSA-Wert würde dort ebenfalls ein Phi-Knoten erzeugt, im Fall von Abhängigkeitskanten geschieht das aber wie oben beschrieben gerade nicht. Stattdessen wird der Phi M-Knoten als Quellknoten der Abhängigkeitskante benutzt. Es könnte auch hinter dem Phi M-Knoten weiter nach einem Schreibzugriff gesucht werden, das würde aber zu dem folgenden Modellierungsfehler bei Schleifen führen, der auch in Abbildung 3.3 dargetellt ist: Befindet sich im Rumpf einer Schleife ein Store-Knoten gefolgt von einem Load-Knoten und die beiden aliasen, so benötigt der Load-Knoten den Speicherwert des Store-Knotens. Gleichzeitig besteht aber auch eine Abhängigkeit zwischen dem Load und dem Store der **nächsten** Schleifeniteration. Eine Abhängigkeitskante vom Store- zum Load-Knoten würde nun die Tatsache, dass die Abhängigkeit über zwei Iterationen der Schleife hinweg besteht, nicht modellie-

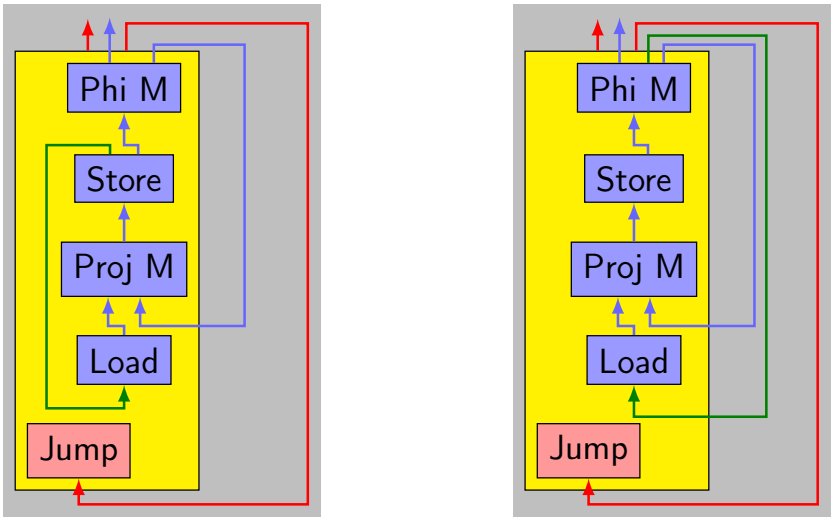


Abbildung 3.3: Um eine zyklische Abhängigkeit (links) zu verhindern muss der Phi-Knoten als Quelle der Abhängigkeitskante benutzt werden (rechts).

ren und zu einer zyklischen Abhängigkeit innerhalb des Schleifenrumpfs führen. In einer solchen Situation ist allerdings garantiert, dass immer ein Phi M-Knoten existiert, der aus dem Speicherwert vor der Schleife und dem aus einer vorangegangenen Schleifeniteration auswählt. Grundsätzlich muss jeder Zyklus in FIRM-Graphen immer mindestens einen Phi-Knoten enthalten. Die Benutzung des Phi M-Knotens als Quelle der Abhängigkeitskante löst das Problem der zyklischen Abhängigkeit, bzw. der Zyklus enthält einen Phi-Knoten. Daher werden Phi M-Knoten in diesem Zusammenhang immer als Schreibzugriffe betrachtet und als Quellknoten für Abhängigkeitskanten benutzt.

Analog zu Phi M-Knoten stellt sich auch bei Sync-Knoten die Frage, ob sie als Quellknoten einer Abhängigkeitskante benutzt werden müssen. In diesem Fall muss allerdings der eigentliche Schreibzugriff, der den Sync-Knoten als Speichereingang hat, verwendet werden. Andernfalls kann es in einigen Fällen dazu kommen, dass durch die Eliminierung gemeinsamer Teilausdrücke (engl. common subexpression elimination, CSE) der Sync-Knoten mit einem anderen Sync-Knoten, der die gleichen Eingänge hat und sich vor dem Load-Knoten befindet, verschmolzen wird, wie es in Abbildung 3.4 dargestellt ist. Das Ergebnis wäre ein Sync-Knoten vor dem Load-Knoten, der alle Abhängigkeitskanten, also auch die zum Load-Knoten hält, wodurch eine zyklische Abhängigkeit entsteht.

3.3 Modellierung von Funktionsaufrufen

In FIRM haben auch Call-Knoten einen Speicherein- und -ausgang. Da für Call-Knoten keine Alias-Abfrage in libFIRM möglich ist, muss davon ausgegangen werden, dass die

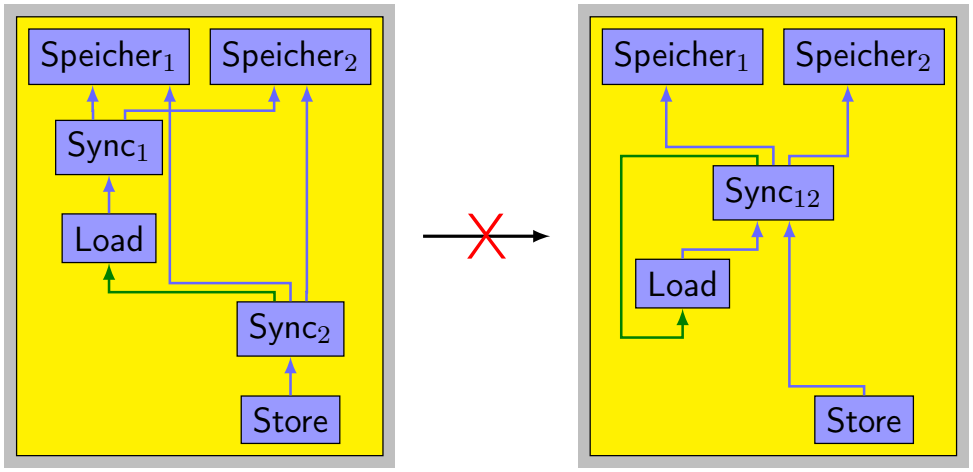


Abbildung 3.4: So nicht: Wird ein Sync-Knoten als Quelle einer Abhängigkeitskante benutzt und dann mit einem anderen Sync-Knoten mit identischen Eingängen verschmolzen, kann eine zyklische Abhängigkeit entstehen.

aufgerufene Funktion auf jeglichen Speicher zugreifen könnte. Call-Knoten brauchen daher die Speicherwerte zu allen Partitionen. Sofern nicht ausgeschlossen werden kann, dass die Funktion auch schreibend auf den Speicher zugreift, definiert der Call-Knoten auch neue Speicherwerte für alle Partitionen. Das ist natürlich der Fall, wenn die Funktion Store-Knoten oder weitere Funktionsaufrufe, die den Speicher verändern könnten enthält, aber auch wenn die aufgerufene Funktion nicht vorliegt und analysiert werden kann. Letzteres ist für Bibliotheksfunktionen oft der Fall. Ist hingegen klar, dass die aufgerufene Funktion nur lesend auf den Speicher zugreifen kann, so wird der Call-Knoten analog zu einem Load-Knoten, der mit allem aliast, behandelt. Das bedeutet, er benötigt alle Speicherwerte, definiert aber keinen neuen und muss mit Abhängigkeitskanten gegen unerlaubtes Verschieben geschützt werden. Dass eine Funktion nur aus dem Speicher liest, kann durch Analyse erkannt werden; manche Programmiersprachen erlauben auch eine Funktion durch ein entsprechendes Schlüsselwort zu kennzeichnen. Für C-Programme stellt eine GCC-Erweiterung zum Beispiel das Schlüsselwort **pure** zur Verfügung.

Allerdings sind Speicher und Termination in Firm wie in Abschnitt 2.1.2 bereits beschrieben vermischt. Greift eine aufgerufene Funktion nicht schreibend auf den Speicher zu und liefert sie keinen Rückgabewert oder wird der Rückgabewert vom Aufrufer verworfen, so könnte der Funktionsaufruf als toter Code entfernt werden. Enthält die aufgerufene Funktion aber eine Endlosschleife, so würde das Entfernen des Aufrufs das Programmverhalten unzulässig verändern. Da Termination nicht entscheidbar ist, muss jede Schleife als potentielle Endlosschleife angesehen werden. Um nun zu verhindern, dass eine Endlosschleife entfernt wird, darf ein Funktionsaufruf nur dann keinen neuen Speicher definieren, wenn die aufgerufene Funktion keine Schleife enthält. Dadurch wird ein Ergebnis der Funktion, nämlich der neue Speicherzustand verwendet und der Funktionsaufruf kann nicht entfernt werden.

Es gibt in Firm noch einige andere Knoten, die sich ähnlich wie Calls verhalten, zum Beispiel die Knoten Alloc und Free, mit denen dynamisch Speicher allokiert und freigegeben wird. Diese Knoten verändern den Speicherzustand und werden wie Call-Knoten behandelt. Weiterhin existieren noch der Builtin-Knoten für Funktionen, die in die Programmiersprache und den Compiler fest eingebaut sind, sowie der ASM-Knoten, der direkt in C-Programme eingebetteten Assembler-Code darstellt. Diese Knoten werden ebenfalls wie Call-Knoten behandelt. Schließlich gibt es in Firm auch noch CopyB-Knoten. Diese Knoten kopieren den Inhalt eines Speicherbereichs, dessen Größe zur Compilezeit bekannt ist, an eine andere Adresse, sind also auch Speicherzugriffe mit Speicherein- und -ausgängen. Sie werden zum Kopieren von Instanzen eines zusammengesetzten Typs benutzt und ebenso konservativ wie Call-Knoten behandelt, die auf allen Speicher zugreifen können, weil eine Alias-Analyse für sie in libFirm nicht möglich ist.

3.4 Vorteile der neuen Modellierung

Die neue Modellierung von Speicherzugriffen ermöglicht es, Alias-Informationen im Programmgraph darzustellen. Dies geschieht jedoch nicht zum Selbstzweck, sondern die Information kann von anderen Phasen des Compilers zur Optimierung des Programms oder schlicht zur Vereinfachung und Beschleunigung der Phase genutzt werden.

Die abgeschwächte Ordnung der Speicherzugriffe bietet dem Scheduling mehr Freiheiten. Das Scheduling ordnet die Knoten jedes Grundblocks linear an, um Maschinencode erzeugen zu können. Die gewonnenen Freiheiten können dabei zur Verfolgung anderer Optimierungsziele verwendet werden. Der in libFirm integrierte Scheduler versucht durch geschickte Befehlsanordnung den Registerdruck möglichst klein zu halten. Dazu werden Knoten, die auf den gleichen Daten arbeiten, nah beieinander angeordnet und so die Lebendigkeitsintervalle einzelner Werte klein gehalten. Es wäre aber auch eine umgekehrte Strategie denkbar. Dabei würden Knoten, die als unabhängig erkannt wurden, hintereinander angeordnet. Superskalare Prozessoren können diese Instruktionen dann parallel ausführen und werden durch diese erhöhte Parallelität auf Instruktionsebene besser ausgelastet.

Darüber hinaus ergibt sich mit der neuen Modellierung eine erhöhte Lokalität im Programmgraphen. Knoten, die auf den gleichen Speicher zugreifen, sind durch Kanten verbunden. Unabhängige Knoten sind dagegen auch im Programmgraph unabhängig. Diese Lokalität kann von Optimierungen, die die Speicherzugriffe betreffen, genutzt werden. So musste früher immer die Kette aus Speicherzugriffen abgesucht werden und dabei mussten Knoten, die nicht aliasen, übersprungen werden, um optimierbare Situationen zu erkennen. Mit der neu gewonnenen Lokalität sind solche Situationen als lokale Muster erkennbar und können besser optimiert werden. Es ergeben sich sogar einige neue Muster, die mit der neuen Modellierung erkannt und optimiert werden können. Neue und verbesserte Optimierungen werden in Kapitel 6 ausführlich beschrieben.

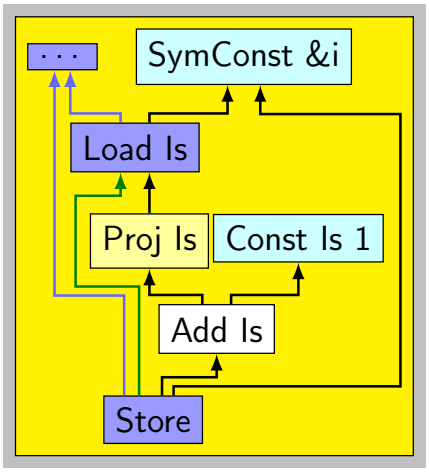


Abbildung 3.5: Das Muster Load-Add-Store kann mit einem einzigen x86-Befehl realisiert werden.

Auch die Instruktionsauswahl kann von der Lokalität profitieren, wenn mehrere Knoten zu einer Maschineninstruktion zusammengefasst werden können. So ist es in x86-Assembler und anderen Assemblersprachen, die einen entsprechenden Addressierungsmodus anbieten, möglich Konstellationen, in denen ein Wert geladen, verändert und sofort an die gleiche Speicherstelle zurück geschrieben wird, in einem einzigen Befehl zu realisieren. Dazu wird die Speicherstelle als ein Operand des Befehls benutzt. Eine solche Konstellation wird durch die neue Lokalität als lokales Muster erkennbar, weil andere Speicherzugriffe zwischen dem Load- und dem Store-Knoten, die nicht mit den Zugriffen auf diese Speicherstelle aliasen, parallel gelegt und damit aus der Konstellation entfernt wurden. Abbildung 3.5 zeigt ein Beispiel für so ein lokales Muster. Hier wird die Variable i um eins inkrementiert und sofort wieder gespeichert. Das lässt sich durch den Befehl **add** $1, i realisieren.

4 Verwandte Arbeiten

Viele optimierende Compiler benutzen Alias-Information um unabhängige Speicherzugriffe zu erkennen und optimieren zu können. So beschreiben etwa Dulong et al. in [DKK99] unter dem Begriff „Memory Disambiguation“ einige im Intel IA-64® Compiler implementierte Techniken zur Gewinnung von Alias-Information und Optimierungen, die dadurch ermöglicht werden. Allerdings wird die Alias-Information nicht in der Zwischendarstellung repräsentiert, sondern muss immer wieder neu berechnet oder in einer eigenen Datenstruktur gespeichert werden. Insgesamt gibt es bisher nur wenige Arbeiten, in denen die Alias-Information tatsächlich in die Zwischendarstellung mit aufgenommen wird.

In [Che00] wird eine Technik beschrieben, bei der zu jeder Funktion in der Zwischensprache eine Liste der Instruktionen, die auf den Speicher zugreifen, und deren Points-To-Mengen als Meta-Informationen gespeichert werden. Aus der Meta-Information jeder Funktion und der Information zu allen möglichen Zielen der vorkommenden Funktionsaufrufe werden dann die Speicherbereiche ermittelt, die von den Instruktionen in der Funktion gelesen oder modifiziert werden können. Daraus lässt sich die Alias-Relation für die Instruktionen ermitteln und das Ergebnis sind so genannte Sync-Arcs, Paare aus zwei Speicherzugriffen, die aliasen. Diese Sync-Arcs werden ebenfalls in den Meta-Informationen zu jeder Funktion gespeichert und müssen von anderen Optimierungen für die Reihenfolge der Instruktionen berücksichtigt werden. Wenn keine Sync-Arcs vorhanden sind, darf die Reihenfolge der Instruktionen verändert werden.

Diego Novillo stellt in [Nov07] „Memory SSA“ vor. Diese Technik wird in der Gnu Compiler Collection (GCC) benutzt. Bei Memory SSA werden ähnlich wie in dieser Arbeit symbolische SSA-Werte für Speicherbereiche eingeführt. Ladende Speicherzugriffe benutzen dann diese SSA-Werte, schreibende Zugriffe konsumieren sie und definieren neue. Für die Aufteilung des Speichers werden verschiedene, unterschiedlich präzise Alternativen beschrieben. Allerdings erfolgt die Darstellung des Zwischencodes in textueller Form, nicht durch Graphen.

In [Ste95] stellt Bjarne Steensgaard die „Assignment Factored SSA Form“ vor. Diese basiert ebenfalls auf der Partitionierung des Speichers. Auf diese Weise wird Alias-Information in Wertabhängigkeitsgraphen (engl. value dependence graph, VDG) repräsentiert. Der Schwerpunkt der Arbeit liegt jedoch nicht auf dem Verfahren zur Partitionierung, sondern auf einer möglichst einfachen und kompakten Darstellungsform der Zwischensprache.

Chow et al. beschreiben in [CCL+96] eine weitere Möglichkeit Alias-Information in SSA-basierten Zwischensprachen zu repräsentieren. Dazu werden zusätzliche Operatoren μ und χ eingeführt, die mögliche Verwendungen und Redefinitionen von SSA-Werten modellieren. Mit diesen zusätzlichen Operatoren werden Speicherzugriffe, die mit diesen SSA-Werten aliasen, annotiert. Um die Anzahl der SSA-Werte zu reduzieren benutzen die Autoren „Zero Versioning“, ein Verfahren, das künstlich erzeugte SSA-Werte wieder zu einem Namen zusammenfasst. Darüber hinaus werden weitere SSA-Namen für indirekte Speicherzugriffe eingeführt und auf diese wird das gleiche Vorgehen angewandt. Diese Konzepte werden dann für die Ableitung von Hashed SSA, einer Zwischensprache in SSA-Form, die auf Global Value Numbering (GVN) basiert, verwendet.

5 Parallelisierungsverfahren

In diesem Kapitel wird ein Verfahren vorgestellt, das FIRM-Graphen in die in Kapitel 3 beschriebene Form bringt. Das heißt, es werden sowohl Alias-Informationen in den Graph eingebracht, als auch die Modellierung von lesenden Speicherzugriffen ohne neu definierten Speicherwert sichergestellt. Die Ausgangsform des FIRM-Graphen ist dabei unerheblich; es dürfen bereits Speicherzugriffe parallelisiert sein und Sync-Knoten vorkommen. Auch dürfen Load-Knoten einen neuen Speicherwert definieren oder nicht. Dadurch kann das Verfahren während des Compiliervorgangs mehrmals angewendet werden, einmal auf den ursprünglichen FIRM-Graphen, der vom Frontend des Compilers erstellt wurde, und danach erneut auf die Ergebnisse anderer Optimierungen.

Das Verfahren wird auf den FIRM-Graph jeder Funktion einzeln angewandt, es arbeitet also intraprozedural. Es wird lediglich, soweit verfügbar, die Information, ob eine durch einen Call-Knoten aufgerufene Funktion schreibend auf den Speicher zugreift oder Schleifen enthält, benutzt, um den Call-Knoten gegebenenfalls als lesenden Zugriff behandeln zu können.

Das Verfahren gliedert sich in drei Phasen:

5.1 Phase I: Speicheroperationen sammeln

Die erste Phase dient lediglich als Vorbereitung der Phasen II und III, um die dort benötigten Informationen zu sammeln. Die Speicheroperationen des Graphen einschließlich Call-Knoten und den Hilfsknoten Phi und Sync werden in dieser Phase einmal besucht. Dazu wird vom initialen Speicherwert, der aus dem Start-Knoten projeziert wird, aus eine modifizierte Breitensuche durchgeführt, bei der nur die Speicherkanten und Abhängigkeitskanten verfolgt werden. Während dieser Breitensuche werden alle Speicheroperationen gesammelt um aus ihnen in Phase II die Partitionierung des Speichers abzuleiten.

Außerdem wird in jedem Grundblock eine gültige Reihenfolge, in der alle Speicheroperationen und Call-Knoten durchgeführt werden können, ermittelt. Diese Reihenfolge wird später für Phase III benötigt. Dazu wird zu jedem Grundblock eine Instruktionsliste angelegt und ein Knoten während der Breitensuche erst bearbeitet, wenn alle seine Vorgänger im gleichen Grundblock bereits besucht wurden. Ist das der Fall, so wird der Knoten in die Instruktionsliste des Grundblocks, zu dem er gehört, eingefügt und seine

Nachfolger werden in der Breitensuche besucht. Ansonsten muss der Knoten später noch einmal von einem anderen Vorgänger aus, der zu diesem Zeitpunkt noch nicht bearbeitet wurde, besucht werden.

Kann zwischen mehreren Durchführungen des Verfahrens die Alias-Information schwächer werden, so müssen beim Einfügen in die Instruktionsliste zusätzlich alle Datenabhängigkeiten berücksichtigt werden. Bei der Alias-Analyse in libFirm ist das leider der Fall. Hier kann es passieren, dass zwei Schreibzugriffe auf unterschiedliche Felder eines zusammengesetzten Typs, die in Firm mit Sel-Knoten modelliert werden, im ersten Durchlauf als unabhängig erkannt und vom Verfahren parallelisiert werden. Später werden die Feldzugriffe durch konkrete Adressrechnung ersetzt. Die Alias-Analyse erkennt die Speicherzugriffe jetzt nicht mehr als unabhängig. Bei einer erneuten Durchführung des Verfahrens kann es dann passieren, dass die Speicherzugriffe in umgekehrter Reihenfolge angeordnet und wegen der schwächeren Alias-Information im Ergebnisgraph nicht parallelisiert werden. Besteht aber gleichzeitig eine Datenabhängigkeit zwischen den beiden Speicherzugriffen, so ergibt sich eine zyklische Abhängigkeit. Kann garantiert werden, dass die Alias-Information nicht schwächer wird, so ist die Reihenfolge in der Instruktionsliste des Grundblockes zwar streng genommen auch falsch, das stellt aber kein Problem dar, weil die Speicherzugriffe wieder parallel angeordnet werden und nur die andere Datenabhängigkeit bestehen bleibt.

5.2 Phase II: Speicherpartitionierung

In Phase II geschieht die eigentliche Aufteilung des Speichers. Die Aufteilung richtet sich dabei nicht nach den tatsächlich im Speicher vorhandenen Daten, sondern nur nach den in Phase I gesammelten Speicherzugriffen und ihrer Alias-Relation. Dadurch wird der Speicher nur so aufgeteilt, dass es auch verschiedene Speicherzugriffe gibt, die einzeln auf die Partitionen zugreifen und somit parallelisiert werden können. Eine Aufteilung darüber hinaus brächte keinen Vorteil. Die Speicherpartitionen werden wegen dieses Ansatzes nur durch die Menge der Speicherzugriffe, die auf sie zugreifen können, repräsentiert, nicht durch die in ihnen enthaltenen Daten. Gleichzeitig ist dadurch die Zuordnung eines Speicherzugriffes zu den Speicherpartitionen, die er benutzt intuitiv klar. Zu beachten ist, dass lediglich der Speicher partitioniert wird, nicht die Menge der Speicherzugriffe. Die Knotenmengen, die den Speicherpartitionen entsprechen, sind nicht disjunkt, wenn ein Speicherzugriff auf mehrere Partitionen zugreifen kann.

Zur Partitionierung des Speichers wurden zwei unterschiedlich mächtige, aber auch unterschiedlich aufwändige Verfahren entwickelt, die im Folgenden vorgestellt werden. Zur Veranschaulichung dieser Verfahren ist der Alias-Graph, wie er in Abschnitt 2.2.2 definiert wurde, hilfreich. Zur Erinnerung, der Alias-Graph besitzt die Speicherzugriffe als Knoten und eine Kante, wenn zwei Knoten aliasen. Dieser Graph dient aber nur zum besseren Verständnis der Verfahren, er wird nie explizit angelegt.

5.2.1 Statische Partitionierung

Eine einfache Art den Speicher zu partitionieren ist die statische Partitionierung. Bei der statischen Partitionierung bleibt die Aufteilung in Speicherpartitionen im gesamten FIRM-Graph gleich, daher kommt auch der Name „statisch“. Nur hinter dem Start-Knoten wird der Speicherwert aufgeteilt und vor jedem Return-Knoten wieder mit einem Sync-Knoten zusammengeführt.

Um an allen Speicheroperationen eine gleichbleibende Partitionierung sicher zu stellen muss der komplette Speicher, auf den eine Speicheroperation zugreifen kann, in einer einzigen Speicherpartition liegen. Sonst wäre vorher ein Sync-Knoten nötig. Daraus ergibt sich auch, dass die Speicherbereiche, auf die zwei Speicheroperationen zugreifen können, immer in der selben Speicherpartition liegen müssen, wenn die Speicheroperationen aliasen. Es muss nämlich zumindest der gemeinsam genutzte Speicher in einer Partition liegen und dann muss sich auch der restliche Speicher für beide Speicheroperationen in dieser Partition befinden. Call-Knoten bilden jedoch eine Ausnahme. Vor Call-Knoten müssen alle Speicherwerte vereinigt und danach wieder aufgetrennt werden, wobei die Auftrennung in genau die gleichen Partitionen wie vor dem Funktionsaufruf erfolgt. Würde man auch an Call-Knoten gleichbleibende Speicherpartitionen verlangen, so würde aller Speicher in eine einzige Partition gezwungen, weil Call-Knoten konservativ so betrachtet werden, als ob sie auf den gesamten Speicher zugreifen können. Dadurch wäre also keine Aufteilung möglich.

Aus der Forderung nach einer gleichbleibenden Partitionierung ergibt sich Algorithmus 1. Die Schleife ab Zeile 2 betrachtet nacheinander jeden FIRM-Knoten und sucht für ihn eine Partition, in der ein anderer Knoten existiert, sodass die beiden aliasen (Zeile 5). Dort wird der Knoten eingefügt (Zeile 7 und 8). Existieren mehrere solcher Partitionen, so werden diese in Zeile 10 und 11 vereinigt. Existiert noch keine solche Partition, so wird in Zeile 12 bis 15 eine neue angelegt, die nur diesen einen Knoten enthält. Das Ergebnis sind gerade die Zusammenhangskomponenten des Alias-Graphen. Die Knoten in verschiedenen Zusammenhangskomponenten sind unabhängig und werden später parallelisiert, indem es für jede Zusammenhangskomponente einen eigenen Speicherwert gibt. Bei dieser Art der Partitionierung wird sowohl der Speicher als auch die Menge der Speicherzugriffe partitioniert. Jeder Zugriff braucht also später nur einen Speicherwert, nämlich den, in dessen Partition er liegt.

Die statische Partitionierung stellt eine recht einfache Aufteilungsmöglichkeit des Speichers dar, die Aufteilung erfolgt aber nur sehr grob, wenn überhaupt. Sobald ein Speicherzugriff vorhanden ist, für den die Alias-Analyse keine einschränkenden Informationen liefert, kann dieser mit allen anderen Zugriffen aliasen. Dadurch ist der Alias-Graph zusammenhängend und es findet keine Aufteilung statt. Abbildung 5.1 zeigt den Alias-Graph zum FIRM-Ausschnitt in Abbildung 3.1, bei dem Store_3 mit Store_1 und Store_2 aliasen konnte, Store_1 und Store_2 untereinander aber nicht. Dies ist so ein Fall, in dem der Alias-Graph zusammenhängend ist und der Speicher nicht aufgeteilt wird; die Paral-

Algorithmus 1: StaticPartitionByAlias

Input : Nodes : Set ⟨FirmNode⟩
Output : Partitions : List ⟨Set ⟨FirmNode⟩⟩

```
1  Partitions ← new List ⟨Set ⟨FirmNode⟩⟩()
2  forall the n ∈ Nodes do
3      partition : Set ⟨FirmNode⟩ ← NIL
4      foreach p ∈ Partitions do
5          if ∃n′ ∈ p : n und n′ aliasen then
6              if partition = NIL then
7                  partition ← p
8                  partition.insert(n)
9              else
10                 partition.addAll(p)
11                 Partitions.delete(p)
12     if partition = NIL then
13         partition ← new Set ⟨FirmNode⟩()
14         partition.insert(n)
15         Partitions.append(partition)
```

Abbildung 5.1: Der Alias-Graph zum Firm-Graph aus Abbildung 3.1.

lelisierung des Firm-Graphen in Abbildung 3.1 wäre also mit statischer Partitionierung nicht möglich.

5.2.2 Flusssensitive Partitionierung

Die flusssensitive Partitionierung teilt den Speicher deutlich aggressiver auf als die statische Partitionierung. Dazu wird die Forderung nach gleichbleibenden Partitionen fallen gelassen. Stattdessen kann die Aufteilung des Speichers sich je nach Position im Steuerfluss unterscheiden. Auch kann jede Speicheroperation bei dieser Art der Partitionierung mehr als einen Speicherwert benötigen. Es wird also nur der Speicher partitioniert, eine Speicheroperation ist in allen Knotenmengen, die die von ihr benutzten Speicherpartitionen reparäsentieren, enthalten. Es sind beliebige Aufspaltungen und Zusammenführungen des Speichers im Funktionsgraph erlaubt.

Für die flusssensitive Partitionierung wird zunächst von einer einzigen großen Partition ausgegangen, die den gesamten Speicher umfasst. Sie wird also durch die Menge aller

in Phase I gesammelten FIRM-Knoten repräsentiert. Diese und alle noch entstehenden Partitionen werden nun iterativ immer wieder aufgeteilt. Eine Aufteilung erfolgt immer dann, wenn in der Knotenmenge zu einer Partition zwei oder mehr Knoten enthalten sind, die nicht miteinander aliasen. In diesem Fall lässt sich nämlich der Speicher, der durch die Knotenmenge repräsentiert wird, so aufteilen, dass in jedem Teil nur der benutzte Speicher von einem solchen Knoten liegt und die Knoten, die nicht miteinander aliasen, parallelisiert werden können. Die Partition wird dazu durch je eine neue Partition zu jedem solchen Knoten ersetzt. Diese neuen Partitionen enthalten den Knoten, der sie verursacht hat, und alle weiteren Knoten der Ausgangspartition, die mit diesem Knoten aliasen. Kann eine Partition nicht mehr aufgeteilt werden, so ist sie fertig und repräsentiert einen Speicherwert im Endergebnis der Partitionierung. Wendet man diese Aufteilungsregel immer wieder an, bis keine Menge mehr aufgeteilt werden kann, so liegen zwei Knoten, die nicht aliasen, garantiert nie in derselben Partition. Das bedeutet, zwei Knoten, die nicht aliasen, benutzen auf jeden Fall unterschiedliche Speicherwerte und können im FIRM-Graph parallelisiert werden. Das Verfahren ist in Algorithmus 2 noch einmal als Pseudocode dargestellt.

Algorithmus 2: FlowSensitivePartitionByAlias

Input : Nodes : Set ⟨FIRMNode⟩
Output : Partitions : Set ⟨Set ⟨FIRMNode⟩⟩
Data : Worklist : Queue ⟨Set ⟨FIRMNode⟩⟩

```
1  Partitions ← new Set ⟨Set ⟨FIRMNode⟩⟩()
2  Worklist.enqueue(Nodes)
3  while ¬ Worklist.isEmpty() do
4      p ← Worklist.dequeue()
5      undividable ← true
6      forall the n ∈ p do
7          if ∃n′ ∈ p : n und n′ aliasen nicht then
8              new_p ← {n} ∪ {m ∈ p \ {n} | m und n aliasen}
9              Worklist.enqueue(new_p)
10             undividable ← false
11     if undividable then
12         Partitions.insert(p)
```

Das Verfahren in Algorithmus 2 kann auch als Fixpunktiteration auf einem Mengensystem von Knotenmengen aufgefasst werden. Die Fixpunktiteration wird mit der Grundmenge aller Knoten aus Phase I initialisiert. In einem Iterationsschritt wird auf jede Menge dieses Mengensystems die Aufteilungsfunktion, wie sie oben textuell beschrieben wurde, als Transformation angewandt, bis keine Aufteilung mehr möglich ist. Dazu muss die Aufteilungsfunktion so erweitert werden, dass sie, wenn keine Aufteilung möglich ist, auf die unveränderte Knotenmenge abbildet. Die folgenden zwei Sätze stellen sicher, dass das Verfahren terminiert und charakterisieren den gefundenen Fixpunkt.

Satz 1. *Algorithmus 2 erreicht einen Fixpunkt und terminiert.*

Beweis. Da die Eingabemenge Nodes endlich ist und alle weiteren Knotenmengen in Zeile 8 als Teilmengen von bereits bestehenden Mengen konstruiert werden, sind alle betrachteten Knotenmengen endlich. Damit terminiert die Schleife in Zeile 6 immer. In Zeile 8 gilt $new\_p \subsetneq p$, denn zumindest der Knoten n', der die Bedingung in Zeile 7 erfüllt, kann nach Konstruktion nicht in new_p enthalten sein. Da p endlich ist, ist also $|new\_p| < |p|$. Somit ist jede Menge entweder nicht weiter aufteilbar und damit fertig (Zeile 11 und 12) oder sie wird durch endlich viele, echt kleinere Teilmengen ersetzt (Zeilen 8 und 9). Enthält eine Knotenmenge p nur ein Element, so ist sie offensichtlich nicht weiter aufteilbar. Da also die Grundmenge endlich ist und die betrachteten Mengen immer kleiner werden, müssen alle Mengen irgendwann nicht mehr weiter aufteilbar sein und Algorithmus 2 terminiert. □

Satz 2. *Der von Algorithmus 2 gefundene Fixpunkt besteht genau aus den maximalen Cliquen des Alias-Graphen.*

Lemma 1. *Die Mengen im gefundenen Fixpunkt sind alles Cliquen des Alias-Graphen.*

Beweis. (durch Widerspruch) Annahme: Eine Menge im Fixpunkt sei keine Clique im Alias-Graph. Dann gäbe es zwei Knoten in der Menge, die im Alias-Graph nicht durch eine Kante verbunden sind und demnach nicht aliasen. Dann wäre die Menge aber aufgeteilt worden und nicht fertig. Also können nur Cliquen im Fixpunkt enthalten sein. □

Lemma 2. *Die Cliquen im Fixpunkt sind alle maximal.*

Beweis. (durch Widerspruch) Annahme: Sei C eine maximale Clique des Alias-Graphen und $C' \subsetneq C$ sei eine nicht-maximale Clique im Fixpunkt. Dann muss irgendwann im Verfahren eine Menge $P \supset C$ so aufgeteilt worden sein, dass für eine der ersetzenden Teilmengen P', aus der C' entsteht, gilt

$$C' \subseteq P' \wedge C \not\subset P'.$$

Dazu muss es einen Knoten $n \in P$ geben, mit dem die Menge P' in Zeile 8 von Algorithmus 2 konstruiert wurde. Für n muss dann gelten

$$\forall n' \in C' : n \text{ und } n' \text{ aliasen} \wedge \forall n' \in C \setminus P' : n \text{ und } n' \text{ aliasen nicht.}$$

Wegen $C \not\subset P'$ muss es gemäß Konstruktionsvorschrift für P' einen Knoten in C geben, sodass n und dieser Knoten nicht aliasen. Da C eine Clique ist, gilt $n \notin C$ und damit auch $n \notin C'$. Damit im Ergebnis der Fixpunktiteration C' übrig bleibt, muss irgendwann noch eine Menge $P'' \supseteq (C' \cup \{n\})$, die aus P' resultiert oder gleich P' ist, so aufgeteilt werden, dass C' und n getrennt werden. Dabei gilt in jedem Fall $P'' \subseteq P'$. Für diese

Aufteilung ist aber ein Knoten $n' \in P''$ nötig, sodass n' und alle Knoten in C' aliasen, n und n' aber nicht. Nach Konstruktion von P' gilt aber

$$\nexists n' \in P' : n \text{ und } n' \text{ aliasen nicht}$$

und wegen $P'' \subseteq P'$ existiert auch kein passender Knoten n' in P''. Also kann C' nicht im Fixpunkt enthalten sein. □

Lemma 3. *Alle maximalen Cliquen sind im Fixpunkt enthalten.*

Beweis. Sei C eine maximale Clique und $P \supseteq C$ eine Menge während des Verfahrens. Falls $P = C$, ist C nicht mehr aufteilbar und im Fixpunkt enthalten. Falls $P \neq C$, so gilt

$$\exists n \in P \setminus C \; \exists n_C \in C : n \text{ und } n_C \text{ aliasen nicht,}$$

denn sonst wäre $C \cup \{n\}$ eine Clique und C wäre nicht maximal. Dann wird P in der nächsten Iteration aufgeteilt und dabei auch ein Knoten n_C gewählt, aus dem die Menge

$$P_C := \{n \in P \mid n \text{ und } n_C \text{ aliasen}\}$$

gebildet wird. Dann gilt auf jeden Fall $C \subseteq P_C$. Damit bleibt immer eine Partition, die C enthält bestehen. Da C auf jeden Fall in der Anfangsmenge der Fixpunktiteration enthalten war, existiert immer eine Knotenmenge die C enthält und C muss auch im Fixpunkt enthalten sein. □

Beweis von Satz 2. Der Satz folgt direkt aus Lemma 1, Lemma 2 und Lemma 3. □

Die Erkenntnis, dass der Fixpunkt gerade aus den maximalen Cliquen im Alias-Graphen besteht, hat zwei wichtige Konsequenzen. Erstens kann es keine Cliquen über eine Zusammenhangskomponente des Alias-Graphen hinaus geben. Deshalb kann man erst das Verfahren für die statische Partitionierung anwenden und dann für die gefundenen Zusammenhangskomponenten eine flusssensitive Partitionierung durchführen. Da die flusssensitive Partitionierung vergleichsweise aufwändig ist, lässt sich damit die Laufzeit der gesamten Partitionierung vermindern.

Zweitens kann es in der Anzahl von Knoten exponentiell viele maximale Cliquen geben [MM65]. Damit ergibt sich eine exponentielle Worst-Case-Laufzeit des Verfahrens. Außerdem muss auch Phase III das exponentiell große Ergebnis weiterverarbeiten. Die daraus resultierende extrem hohe Laufzeit konnte bereits bei einigen Testfällen aus der SPECint2000 beobachtet werden. Der exponentiell große Fall beschränkt sich also nicht nur auf einige konstruierte Spezialfälle, sondern tritt auch bei praxisnahen Programmen auf. In [BK73] wurde der Bron-Kerbosch-Algorithmus zur Berechnung aller maximalen Cliquen eines Graphen beschrieben, der Branch-And-Bound einsetzt um die Laufzeit zu senken. Die exponentielle Laufzeit lässt sich allerdings durch Verbesserungen des Verfahrens nicht beseitigen, weil die Ausgabe bereits exponentiell groß sein kann.

Das hier vorgestellte Verfahren kann aber im Gegensatz zum Bron-Kerbosch-Algorithmus vorzeitig abgebrochen werden. Dazu wird die Erkenntnis aus Kapitel 3 genutzt, dass zu einer Partitionierung auch jede andere Partitionierung gültig ist, die mehrere Partitionen in einer zusammenfasst. Da das Verfahren gerade so angelegt ist, dass nacheinander je eine Partition weiter aufgeteilt wird, darf also auch jede Partition aus der Arbeitsliste unzerteilt in die Menge der endgültigen Partitionen aufgenommen werden, auch wenn sie noch weiter aufgeteilt werden könnte. Algorithmus 3 zeigt eine modifizierte Variante des Algorithmus zur flusssensitiven Partitionierung. In Zeile 12 wird nun zusätzlich die Methode `Accept` aufgerufen, die eine geeignete Heuristik implementiert um eine Aufteilung zu akzeptieren oder abzulehnen. Die Heuristik kann selbstverständlich in die Berechnung der Aufteilung integriert werden und sollte dies auch, wenn sie einen früheren Abbruch etwa bereits in der Schleife ab Zeile 7 erlaubt.

Algorithmus 3: FlowSensitivePartitionByAlias2

Input : Nodes : Set ⟨FirmNode⟩
Output : Partitions : Set ⟨Set ⟨FirmNode⟩⟩
Data : Worklist : Queue ⟨Set ⟨FirmNode⟩⟩

```
1  Partitions ← new Set ⟨Set ⟨FirmNode⟩⟩()
2  Worklist.enqueue(Nodes)
3  while ¬ Worklist.isEmpty() do
4      p ← Worklist.dequeue()
5      undividable ← true
6      proposed_partitions ← new Set ⟨Set ⟨FirmNode⟩⟩()
7      forall the n ∈ p do
8          if ∃n′ ∈ p : n und n′ aliasen nicht then
9              new_p ← {n} ∪ {m ∈ p \ {n} | m und n aliasen}
10             proposed_partitions.insert(new_p)
11             undividable ← false
12     if undividable ∨ ¬ Accept(p, proposed_partitions) then
13         Partitions.insert(p)
14     else
15         forall the p′ ∈ proposed_partitions do
16             Worklist.enqueue(p′)
```

Als Heuristik kann zum Beispiel einfach die Gesamtanzahl der Partitionen begrenzt werden. Hat die Anzahl der fertigen Partitionen und der noch zu betrachtenden Elemente in der Arbeitsliste eine gewisse Grenze erreicht, so wird keine weitere Aufteilung mehr vorgenommen und der Inhalt der Arbeitsliste den fertigen Partitionen hinzugefügt. Allerdings lieferte diese Heuristik weder mit einer konstanten Schranke noch mit einer Beschränkung in Abhängigkeit der Gesamtanzahl von Speicheroperationen zufriedenstellende Ergebnisse. Stattdessen wird in der im Rahmen dieser Arbeit implementierten

Version des Verfahrens die Überlappung der Knotenmengen als Qualitätsmaß herangezogen. Wie bereits erwähnt wird nur der Speicher partitioniert. Die Knotenmengen, die die einzelnen Partitionen repräsentieren, müssen nicht disjunkt sein. Ein Knoten braucht später die Speicherwerte zu allen Partitionen, in deren Knotenmengen er enthalten ist. Eine geringe Überlappung der Knotenmengen, in die aufgeteilt wird, entspricht daher einem großen Parallelisierungspotenzial im FIRM-Graphen, weil die Speicheroperationen nur selten auf gemeinsamen Speicher zugreifen. Die Heuristik akzeptiert deshalb eine Aufteilung genau dann, wenn die Mengen, in die aufgeteilt wird, zusammen höchstens fünfmal so groß sind wie die Größe der ursprünglichen Menge. Das bedeutet, jeder Knoten kann durchschnittlich in höchstens fünf neuen Partitionen liegen. Die Zahl fünf ist dabei natürlich recht willkürlich gewählt, hat sich aber in Experimenten bewährt.

Bei der Implementierung des Verfahrens ist noch ein anderer Aspekt entscheidend für die Laufzeit, nämlich die Duplikatfreiheit der Mengen. Während des Verfahrens kann es vorkommen, dass die gleiche Knotenmenge mehrmals entsteht. Das ist zum Beispiel der Fall, wenn zwei Knoten auf exakt denselben Speicher zugreifen und deshalb auch mit den gleichen Knoten aliasen. Wird mit einem solchen Knoten in Zeile 9 von Algorithmus 3 eine neue Knotenmenge konstruiert, dann wird die gleiche Knotenmenge auch noch einmal mit dem anderen Knoten gebildet und der gesamte weitere Aufteilungsprozess doppelt durchgeführt. In Algorithmus 3 ist *proposed_partitions* als Menge organisiert und das Duplikat entfällt einfach. Bei der Implementierung kann man den zweiten Knoten markieren, wenn er zur ersten Knotenmenge hinzugefügt wird, sodass mit ihm keine Knotenmenge mehr gebildet wird.

5.2.3 Zusammenhang zwischen den Partitionierungsvarianten

Das Wissen, dass die maximalen Cliquen des Alias-Graphen eine geeignete Partitionierung des Speichers darstellen, lässt auch eine neue Interpretation der statischen Partitionierung zu. Mit dieser neuen Interpretation wird ein direkter Zusammenhang mit der flusssensitiven Partitionierung erkennbar.

Aus Abschnitt 2.2.2 ist bekannt, dass die Alias-Relation im Allgemeinen nicht transitiv ist. Wäre sie transitiv, so wären die Zusammenhangskomponenten und die maximalen Cliquen im Alias-Graph identisch, weil dieser ungerichtet ist. Mit einer transitiven Alias-Relation wäre also die Berechnung einer Speicherpartitionierung in Form der maximalen Cliquen deutlich einfacher und das Ergebnis könnte auch nicht mehr exponentielle Größe haben.

Das Verfahren zur statischen Partitionierung lässt sich nun derart interpretieren, dass es erst zur transitiven Hülle des Alias-Graphen übergeht und dann die maximalen Cliquen berechnet. Durch den Übergang zur transitiven Hülle verliert die erhaltene Speicherpartitionierung an Genauigkeit. Gleichzeitig kann sie effizienter berechnet werden, weil die maximalen Cliquen der transitiven Hülle gerade die Zusammenhangskomponenten des

originalen Alias-Graphen sind. Aus diesem Blickwinkel stellt die statische Partitionierung eine abgeschwächte Variante der flusssensitiven Partitionierung dar.

5.3 Phase III: Bestimmen der neuen Abhängigkeiten

In Phase III werden für alle Speicherzugriffe die neuen Speicherwerte und Abhängigkeitskanten entsprechend zur Partitionierung in der vorangegangenen Phase ermittelt. Dazu wird eine modifizierte Breitensuche auf dem Steuerflussgraph durchgeführt, die jeden Grundblock mindestens einmal besucht. Zusätzlich muss sichergestellt sein, dass, nachdem ein Grundblock das erste Mal besucht wurde, alle seine Nachfolger noch einmal besucht werden. Algorithmus 4 zeigt das Grundgerüst für diese Phase.

Algorithmus 4: DetermineNewDependencies

```
   Input : Partitions : List ⟨Set ⟨FirmNode⟩⟩
   Data : Worklist : Queue ⟨Block⟩
1  Worklist.enqueue(StartBlock)
2  while ¬ Worklist.isEmpty() do
3      block ← Worklist.dequeue()
4      if ¬block.visited then
5          InitBlockMems(block, Partitions)
6          BuildIntraBlockMems(block, Partitions)
7          block.visited ← true
           // add block's successors to Worklist
8          forall the s ∈ block.succs do
9              Worklist.enqueue(s)
10     else
11         CompletePhis(block)
```

Wird ein Block das erste Mal besucht, werden als erstes die initialen Speicherwerte, die ihn aus seinen Vorgängerblöcken erreichen, durch Algorithmus 5 ermittelt. Dazu besitzt jeder Grundblock ein Array *mems*, das zu jeder Partition den Knoten m speichert, der den zugehörigen Speicherwert zuletzt definiert hat, sowie die Menge l aller Lesezugriffe, die noch vor der nächsten Redefinition des Speicherwertes passieren müssen. Sie werden als „offene“ Lesezugriffe bezeichnet. Der Startblock besitzt keine Vorgängerblöcke, bei ihm wird dieses Array stattdessen mit dem initialen Speicherwert der Funktion initialisiert; offene Lesezugriffe gibt es noch nicht (Zeile 1 bis 3). Für alle anderen Grundblöcke ergeben sich die initialen Speicherwerte gerade aus den finalen Speicherwerten der Voränger. Besitzt ein Grundblock nur einen Voränger, so werden einfach dessen finale Speicherwerte übernommen (Zeile 6 und 7). Dabei muss der Grundblock über diesen einzigen

Vorgänger in der Breitensuche erreicht worden sein, der Vorgänger ist also auf jeden Fall schon bearbeitet. Besitzt ein Grundblock mehr als einen Vorgänger, so wird in Zeile 9 bis 16 ein Phi-Knoten eingefügt, der den initialen Speicherwert in diesem Block definiert. Der Phi-Knoten wählt aus den finalen Speicherwerten der Vorgängerblöcke den richtigen aus. Außerdem hält er die Abhängigkeitskanten zu allen offenen Lesezugriffen (vgl. Abschnitt 3.2). Wurde ein Vorgängerblock noch nicht bearbeitet, so bleibt die zugehörige Position am Phi-Knoten zunächst leer. Um den Phi-Knoten später fertig stellen zu können, speichert er in Zeile 15 den Index der Speicherpartition, zu deren Speicherwert er gehört, und er wird in Zeile 16 im Block gespeichert.

Algorithmus 5: InitBlockMems

Input : Block : FirmBlock
Partitions : List ⟨Set ⟨FirmNode⟩⟩

```
1  if Block = StartBlock then
2      for i ← 1 to Partitions.length() do
3          Block.mems[i] ← (InitialMem, ∅)
4  else if Block ≠ EndBlock then
5      for i ← 1 to Partitions.length() do
6          if Block.pred_count == 1 then
7              Block.mems[i] ← Block.pred[1].mems[i]
8          else
9              phi ← new Phi(Block.pred_count)
10             for j ← 1 to Block.pred_count do
11                 if Block.pred[j].visited then
12                     phi.pred[j] ← Block.pred[j].mems[i].m
13                     phi.deps.addAll(Block.pred[j].mems[i].l)
14             Block.mems[i] ← (phi, ∅)
15             phi.mem_value ← i
16             Block.new_phis.insert(phi)
```

Nachdem die initialen Speicherwerte eines Grundblocks bestimmt sind, werden von Algorithmus 6 die Speicherabhängigkeiten innerhalb des Grundblocks berechnet. Dazu wird die Instruktionsliste des Blockes aus Phase I durchlaufen. Für jeden Speicherzugriff werden mögliche alte Abhängigkeitskanten entfernt. Dann wird geprüft, auf welche Speicherpartitionen er zugreift, das heißt, welche Speicherwerte er benutzt. Entsprechend der Modellierung aus Kapitel 3 benutzen Call- und Return-Knoten alle Speicherwerte, andere Knoten benutzen genau die, in deren Knotenmenge sie enthalten sind. Alle benutzten Speicherwerte werden dann dem neuen Speicherwert *mem* hinzugefügt, was gegebenenfalls einen Sync-Knoten verursacht. Lesende Speicherzugriffe, also Load-Knoten und Call-Knoten, die nur lesend auf den Speicher zugreifen, werden in Zeile 7 und 8 den offenen Lesezugriffen des Speicherwertes hinzugefügt. Schreibende Speicherzugriffe erhal-

ten in Zeile 10 Abhängigkeitskanten zu alle offenen Lesezugriffen. Außerdem definieren sie, genauer der Proj M-Knoten, der an ihnen hängt, den Speicherwert neu und werden in Zeile 12 in *mems* gespeichert. In Zeile 13 wird der neu berechnete Speicherwert *mem* am Knoten n eingetragen. Am Ende enthält das Array *mems* des Grundblocks die finalen Speicherwerte, die die Nachfolgeblöcke erreichen.

Algorithmus 6: BuildIntraBlockMems

Input : Block : FirmBlock
Partitions : List ⟨Set ⟨FirmNode⟩⟩

```
1  foreach n ∈ Block.instructions do
2      mem ← NIL
3      n.deps.clear()
4      for i ← 1 to Partitions.length() do
5          if n.type = Call ∨ n.type = Return ∨ n ∈ Partions[i] then
6              mem ← merge(mem, Block.mems[i].m)    // might create a Sync
7              if n.type = Load ∨ n.type = readonly-Call then
8                  Block.mems[i].l.insert(n)
9              else
10                 n.deps.addAll(Block.mems[i].l)
11                 Block.mems[i].l ← ∅
12                 Block.mems[i].m ← n.proj_m
13     n.mem ← mem
```

Wurde ein Grundblock in der Breitensuche schon einmal besucht, aber es waren zu diesem Zeitpunkt noch nicht alle seine Vorgängerblöcke bearbeitet, so müssen nur noch die leer gebliebenen Eingänge an den Phi-Knoten fertig gestellt werden. Dazu untersucht Algorithmus 7, wenn ein Grundblock noch einmal besucht wird, alle Eingänge von Phi-Knoten dieses Grundblocks. Wird ein leerer Eingang gefunden und ist der zugehörige Vorgängerblock mittlerweile besucht worden, so werden der finale Speicherwert, zu dem der Phi-Knoten gemäß seines Attributs *mem_value* gehört, und alle offenen Lesezugriffe auf diesen Speicherwert ergänzt.

Algorithmus 7: CompletePhis

Input : Block : FirmBlock

```
1  forall the phi ∈ Block.new_phis do
2      for j ← 1 to Block.pred_count do
3          if phi.pred[j] = NIL ∧ Block.pred[j].visited then
4              phi.pred[j] ← Block.pred[j].mems[phi.mem_value].m
5              phi.deps.addAll(Block.pred[j].mems[phi.mem_value].l)
```

Die Phase III führt einen SSA-Aufbau für die neuen Speicherwerte durch. Dabei werden in jedem Grundblock, an dem Steuerfluss zusammenläuft, vorläufige Phi-Knoten erzeugt, die später wieder entfallen können, etwa weil alle ihre Eingänge gleich sind oder weil sie eine sinnlose Schleife bilden, indem sie nur sich selbst und einen anderen Knoten als Eingang benutzen. Auch können identische Phi-Knoten für unterschiedliche Speicherwerte erzeugt werden, wenn diese Speicherwerte alle von denselben Knoten definiert werden. Solche überflüssigen Phi-Knoten können während dieser Phase im Allgemeinen noch nicht erkannt werden, weil einige Vorgängerblöcke noch nicht bearbeitet sind und die Eingänge der Phi-Knoten deshalb noch nicht alle bekannt sind, wenn sie erzeugt werden.

5.4 Einordnung in den Compiliervorgang

Während des Compiliervorgangs findet nacheinander eine ganze Reihe von Optimierungen statt, die den Firm-Graph transformieren. Es muss also eine geeignete Position für die in dieser Arbeit vorgestellte neue Optimierung gefunden werden.

Um die durch die Parallelisierung gewonnenen Freiheiten zum Scheduling nutzen zu können, reicht es, wenn die neue Optimierung erst kurz vor dem Ende des Compiliervorgangs stattfindet. Auch um die Lokalität bei der Instruktionsauswahl ausnutzen zu können ist eine Platzierung am Ende ausreichend. Darüber hinaus sollen aber auch möglichst viele andere Optimierungen von der im Firm-Graph repräsentierten Alias-Information und der veränderten Modellierung von Lesezugriffen profitieren. Daher sollte diese Optimierung so früh wie möglich stattfinden.

Zusätzlich sollte die Optimierung ein zweites Mal nach dem Inlining von Methoden erfolgen. Dadurch ist oft ein größeres Parallelisierungspotenzial vorhanden, weil einige Call-Knoten weggefallen sind, die vorher nur vereinfacht behandelt wurden, als ob sie auf allen Speicher zugreifen können. Stattdessen werden nach dem Inlining die Speicherzugriffe in der aufgerufenen Methode für die Partitionierung des Speichers mitberücksichtigt und mit parallelisiert.

Im späteren Verlauf des Compiliervorgangs werden noch die CopyB-Knoten entweder durch eine Folge von Load- und Store-Instruktionen oder durch einen Aufruf einer Systemfunktion ersetzt. Die hier vorgestellte Optimierung sollte erst nach der Ersetzung der CopyB-Knoten erfolgen um die neu enstehenden Load- und Store-Knoten mit einbeziehen zu können, anstatt von der konservativen Betrachtung der CopyB-Knoten, als würden sie auf allen Speicher zugreifen, bei der Parallelisierung behindert zu werden. Die Ersetzung der CopyB-Knoten hängt allerdings von einigen Parametern der Zielplattform, wie zum Beispiel der Wortbreite, ab und wird daher erst sehr spät vorgenommen. Die neue Optimierung dahinter zu platzieren würde deshalb dem oben formulierten Ziel, sie möglichst früh durchzuführen, widersprechen und sie verbleibt daher vor der Ersetzung der CopyB-Knoten.

Wegen des SSA-Aufbaus mit vorläufigen Phi-Knoten in Phase III müssen nach dieser Optimierung noch eventuelle überflüssige Phi-Knoten entfernt werden. Das sollte direkt nach dieser Graphtransformation geschehen, um den Firm-Graph klein zu halten und weitere Optimierungen nicht mit diesen überflüssigen Knoten zu belasten. Dazu werden einige lokale Optimierungen durchgeführt, die unter anderem einzelne Phi-Knoten, bei denen alle Eingänge gleich sind, entfernen und mehrere Phi-Knoten mit identischen Eingängen zu einem verschmelzen. Außerdem wird die Entfernung von Phi-Zyklen durchgeführt. Solche Phi-Zyklen lassen einen Wert unverändert durch eine Schleife und wieder aus ihr heraus gelangen. Diese Phi-Knoten sind dann überflüssig, weil alle ihre Benutzer auch den Wert vor der Schleife benutzen können.

5.5 Abhängigkeitskanten in anderen Optimierungen

Bisher waren Abhängigkeitskanten in libFirm dem Backend vorbehalten. Da sie von der neuen Optimierung nun sehr früh in den Firm-Graph eingebracht werden, sind Anpassungen an vielen bereits implementierten Optimierungen nötig, sodass diese die Abhängigkeitskanten berücksichtigen. Dabei müssen Abhängigkeitskanten erstens bei der Erkennung von optimierbaren Situationen beachtet und zweitens auch beim anschließenden Umbau des Firm-Graphen mit transformiert werden. So müssen etwa alle Optimierungen, die einen Grundblock in zwei aufeinander folgende Grundblöcke aufteilen, sicherstellen, dass, wenn ein Knoten im ersten der beiden neuen Blöcke platziert wird, auch alle Lesezugriffe, zu denen er Abhängigkeitskanten hat, in diesen Grundblock kommen, auch wenn der gelesene Wert erst später gebraucht wird.

Durch einige Optimierungen kann es auch passieren, dass der Quellknoten einer Abhängigkeitskante entfällt. Dann muss die Kante nach – in Steuer- und Datenflussrichtung – hinten verschoben und dazu entgegen der üblichen Kanten-Richtung durch den Firm-Graph navigiert werden. Dabei darf die Abhängigkeitskante nicht an einem Sync-Knoten angebracht werden (vgl. Abschnitt 3.2). Stattdessen ist es nötig die Abhängigkeitskante noch weiter zu den Benutzern des Sync-Knotens zu verschieben. Der Benutzer kann aber nicht wieder ein Sync-Knoten sein, weil Sync-Knoten als Eingänge von anderen Sync-Knoten immer ersetzt werden, indem der zweite Sync-Knoten die Eingänge des ersten erhält.

Weiterhin kann es passieren, dass ein Knoten eliminiert werden kann, der den von einem Load-Knoten geladenen Wert nur indirekt benutzt hat. Gibt es keine weiteren Benutzer, so wird der Load-Knoten nicht mehr gebraucht, ohne dass dies unmittelbar erkannt wird, weil sich zwischen dem Load und dem zuerst weggefallenen Knoten weitere Knoten befinden. In einem solchen Fall bleibt der Load-Knoten als toter Code erhalten, weil er noch von einer Abhängigkeitskante festgehalten wird. Diese toten Abhängigkeiten müssen vor Transformationen, die damit nicht funktionieren, und einmal am Ende der Compileroptimierungen in einer zusätzlichen Graph-Transformation entfernt werden.

In einigen Optimierungen führen auch die Aufgabe der SSA-Form von Abhängigkeitskanten und die daraus resultierende Modellierungsentscheidung, dass sie an Phi M-Knoten angebracht werden, zu Problemen. Ein Beispiel dafür ist die Optimierung „Jump-Threading“[1], die für den Umbau des FIRM-Graphen die Information, über welchen Vorgängerblock eine Kante einen Phi-Knoten erreicht, benötigt. Bei SSA-Werten ist jede eingehende Kante des Phi-Knotens genau einem Vorgängerblock zugeordnet, Abhängigkeitskanten hängen hingegen einfach mit an den Phi M-Knoten. Die Zugehörigkeit einer solchen Abhängigkeitskante zu einem bestimmten Eingang des Phi-Knotens bzw. dem zugehörigen Vorgänger-Grundblock kann aber ermittelt werden, indem die Dateneingänge des Phi M-Knotens mit dem Speichereingang des Load- oder Call-Knotens, zu dem die Abhängigkeitskante führt, verglichen werden. Die durch die Abhängigkeitskante ausgedrückte Reihenfolge-Abhängigkeit besteht immer über den Ausführungspfad, dessen zugehöriger Phi-Eingang mit dem Speichereingang des Load- oder Call-Knotens übereinstimmt.

Auch für die Code-Platzierung stellt die fehlende SSA-Eigenschaft der Abhängigkeitskanten ein Problem dar. Diese Optimierung versucht Knoten in geeignetere Grundblöcke zu verschieben. Dabei müssen nun auch Abhängigkeitskanten berücksichtigt werden. Durch die fehlende SSA-Eigenschaft wird es aber schwieriger einen passenden Grundblock für einen Knoten zu finden, weil der Zielknoten einer Abhängigkeitskante den Quellknoten nicht zu dominieren braucht, was mithilfe des Dominanzbaumes leicht zu prüfen wäre. Stattdessen muss sichergestellt sein, dass der Zielknoten auf keinem Ausführungspfad außer über Schleifen vom Quellknoten erreicht werden kann. Abhängigkeitskanten stellen deshalb bei der Suche nach einem neuen Grundblock einen Sonderfall dar und müssen extra behandelt werden.

Selbstverständlich müssen Abhängigkeitskanten auch in der Optimierung von Speicherzugriffen beachtet werden. Hier ändert sich aber noch einiges mehr und es kommen sogar neue Optimierregeln hinzu. Deshalb wird diese Optimierung im nächsten Kapitel ausführlicher behandelt.

[1] JumpThreading versucht bedingte Sprünge schon zur Compilezeit auszuwerten und dadurch einzusparen. So braucht bei klassischen Zählschleifen die Abbruchbedingung oft nicht vor dem ersten Schleifendurchlauf geprüft werden, weil der initiale Wert und die obere Schranke der Zählvariable als Konstanten bekannt sind.

6 Optimierungen

Die Optimierung von Speicherzugriffen ist der wichtigste Nutzer von Alias-Information. Durch die Repräsentation dieser Information im Programmgraph lassen sich viele Optimierungen, die Speicherzugriffe betreffen, vereinfachen und verbessern. Mehrere optimierbare Situationen werden zu lokalen Mustern, die effizient erkannt werden können, anstatt größere Abschnitte des Firm-Graphen unter Berücksichtigung externer Alias-Information untersuchen zu müssen. Zusätzlich erlaubt die neue Modellierung von Lesezugriffen ohne Speicherausgang eine leichtere Eliminierung überflüssiger Load-Knoten. Dadurch ließ sich die vorhandene Optimierung von Speicherzugriffen in libFirm im Rahmen dieser Arbeit effizienter machen und sogar um einige neue Muster erweitern.

Es ist allerdings zu beachten, dass sich durch viele dieser Optimierungen der Registerdruck erhöht, wenn ein Load-Knoten entfernt wird. Dann muss nämlich der frühere Wert, der jetzt anstelle des geladenen Wertes benutzt werden soll, bis zu dieser Benutzung im Prozessor gehalten werden.

6.1 Laden eines gerade geschriebenen Wertes

Wird in den Speicher geschrieben und direkt danach von der gleichen Adresse wieder geladen, so liefert der Lesezugriff in jedem Fall den gerade geschriebenen Wert. Dieser Lesezugriff kann dann eingespart werden und stattdessen lässt sich der Wert, der in den Speicher geschrieben wurde, weiter benutzen, in der Hoffnung, dass er noch in einem Register des Prozessors verblieben ist und nicht in den Hauptspeicher ausgelagert werden musste. Sonst erhöht sich der Registerdruck, weil der gespeicherte Wert auch nach dem Schreibzugriff noch lebendig bleibt.

In Firm sieht diese Situation so aus wie in Abbildung 6.1 dargestellt: Ein Load-Knoten benutzt einen Store-Knoten als Speichereingang und beide greifen auf dieselbe Adresse zu. Dann können alle Verwender des geladenen Wertes stattdessen das Original, das gespeichert wurde, benutzen. Der Load-Knoten kann eliminiert und damit ein Speicherzugriff eingespart werden.

Ohne dass die Alias-Information in den Firm-Graph eingebracht wurde, ist diese Situation zwar auch optimierbar. Allerdings kann es dann zwischen dem Load- und dem Store-Knoten beliebig viele weitere Speicherzugriffe geben, die aber nicht mit dem Load- oder

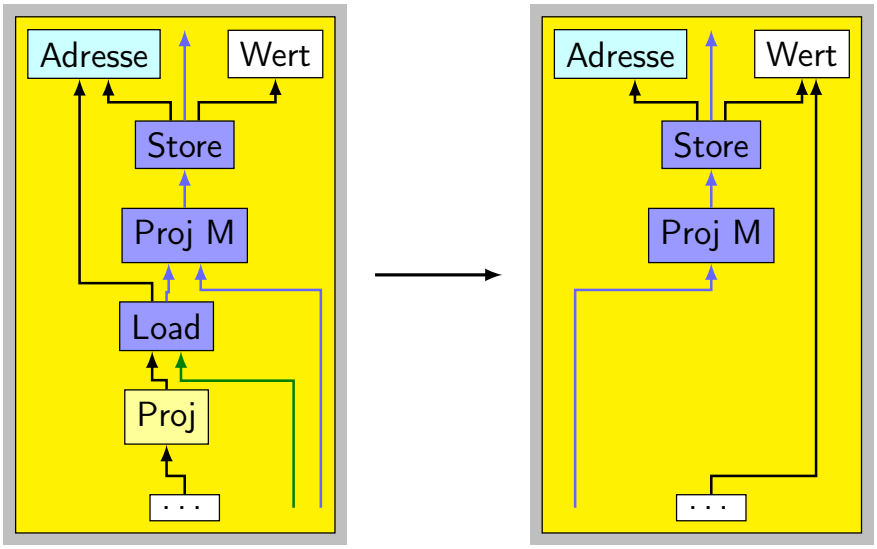


Abbildung 6.1: Anstatt den Wert neu aus dem Speicher zu laden, kann hinter dem Store-Knoten das Original weiter verwendet werden.

Store-Knoten aliasen dürfen. Diese Kette von Speicherzugriffen muss komplett abgesucht werden und dabei müssen alle Knoten überprüft werden, ob sie mit dem Load- oder Store-Knoten aliasen. Durch die Repräsentation der Alias-Information im Programmgraph ergibt sich eine erhöhte Lokalität von Speicherzugriffen, die aliasen, von der diese Optimierung profitiert. Der Load-Knoten benutzt direkt den einzigen, für diese Optimierung in Frage kommenden Store-Knoten als Speicherwert. Weitere Knoten, die weder mit dem Load- noch mit dem Store-Knoten aliasen, liegen nicht mehr zwischen Load und Store, sondern parallel. Um diese Situation zu erkennen brauchen deshalb nur vier Knoten – Load, Proj M, Store und der Adressknoten – betrachtet werden.

6.2 Überschreiben eines gerade geschriebenen Wertes

Auch bei zwei aufeinander folgenden Schreibzugriffen gibt es eine Situation, die mithilfe der Alias-Information im Firm-Graphen leichter optimiert werden kann. Wenn der zweite Schreibzugriff den Wert, den der erste geschrieben hat, wieder überschreibt, ohne dass dieser Wert zwischenzeitlich geladen werden kann, so ist der erste Schreibzugriff tot, das heißt, er trägt nichts zum Ergebnis der Funktion bei. Der tote Schreibzugriff kann dann eliminiert werden.

Abbildung 6.2 zeigt diese Situation in Firm. Sofern Wert 2 mindestens so viel Speicherplatz braucht wie Wert 1 und niemand sonst den Speicherwert, der vom ersten Store-Knoten definiert wurde, benutzt, kann dieser Knoten entfernt werden. Hat darüber hinaus der Knoten Wert 1 keine weiteren Benutzer, so werden er und alle Knoten, die zu

seiner Berechnung nötig waren, durch diese Optimierung ebenfalls überflüssig und können gelöscht werden.

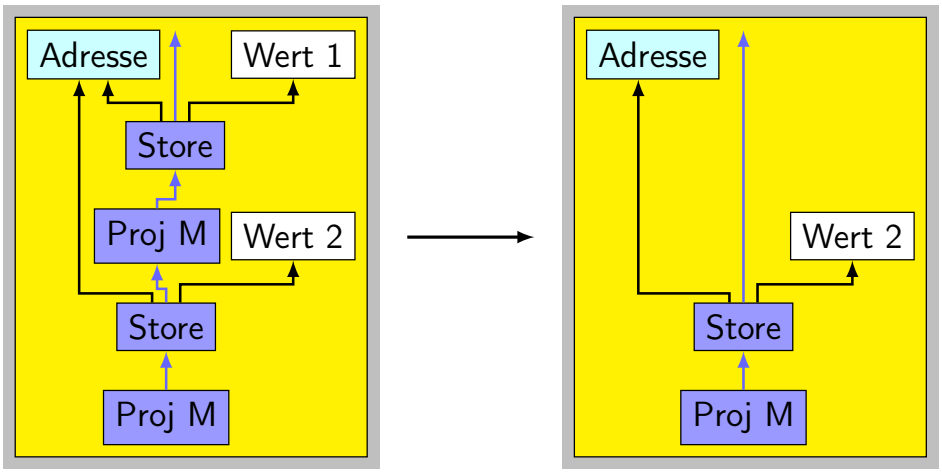


Abbildung 6.2: Wird ein Wert in den Speicher geschrieben und sofort wieder überschrieben, so kann der erste Store-Knoten ersatzlos gestrichen werden.

Der Vorteil durch die Repräsentation der Alias-Information im Firm-Graph ist hier der gleiche wie bei der Optimierung, die in Abschnitt 6.1 beschrieben wurde, nämlich die stärkere Lokalität von Speicherzugriffen, die miteinander aliasen. Mögliche weitere Speicherzugriffe, die zwischen den beiden Store-Knoten lagen, aber nicht mit ihnen aliasen können, liegen nun parallel und brauchen nicht betrachtet und auf mögliches Aliasing geprüft werden.

6.3 Neuladen eines bereits geladenen Wertes

Wird zweimal nacheinander von der gleichen Adresse geladen, ohne dass der Wert an dieser Adresse zwischendurch überschrieben wird, kann der zweite Lesezugriff elimiert und stattdessen der bereits geladene Wert noch einmal benutzt werden. In Firm stellt sich diese Situation in Form von zwei Load-Knoten auf demselben Ausführungspfad dar, die denselben Knoten als Adresse benutzen.

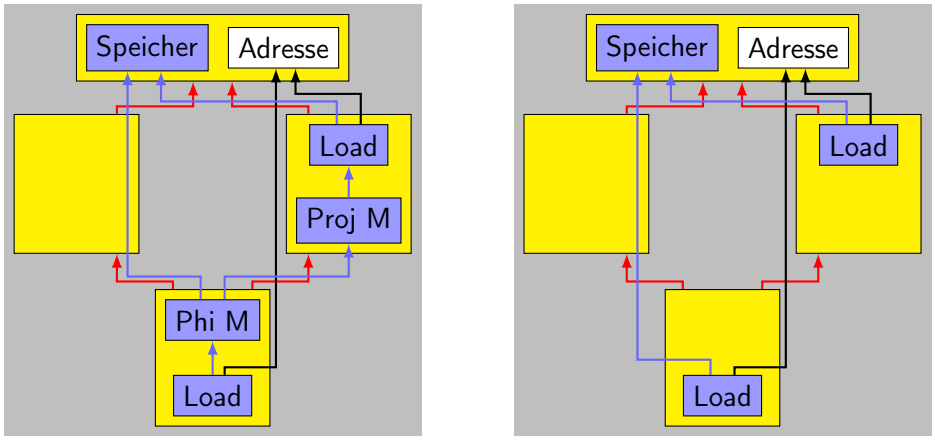
Diese Optimierung profitiert einerseits wieder von der Repräsentation der Alias-Information im Firm-Graph, weil Store-Knoten, die nicht mit den Load-Knoten aliasen, nicht mehr zwischen diesen sondern parallel liegen. Zusätzlich wird diese Optimierregel auch durch die neue Modellierung der Load-Knoten ohne Speicherausgang vereinfacht. Dadurch benutzen zwei in Frage kommende Load-Knoten nicht nur denselben Adressknoten, sondern auch denselben Speicherwert und nicht der zweite Knoten den Speicherwert des ersten. Somit sind die Knoten wirklich gleich und können, wenn sie im selben Grundblock stehen, von der CSE erkannt und verschmolzen werden. Aber auch wenn die Load-

Knoten in unterschiedlichen Grundblöcken stehen, lassen sie sich von ihrem gemeinsamen Speicherwert aus finden, also als lokales Muster und nicht durch Absuchen einer Kette von Speicherzugriffen. Dominiert nun der Grundblock des einen Load-Knotens den des anderen, so kann auch hier der andere Load-Knoten eleminiert werden. Die Einsparung dieses Speicherzugriffs erhöht dabei allerdings wieder den Registerdruck, weil der geladene Wert bis zu seiner letzten Verwendung im Prozessor gehalten werden muss.

6.4 Eliminierung partieller Redundanzen

Partielle Redundanzen sind Berechnungen, die auf einigen Ausführungspfaden des Programms mehrfach durchgeführt werden, aber nicht auf allen. Die Eliminierung partieller Redundanzen versucht solche mehrfachen Berechnungen zu vermeiden. Eine Implementierung dieser Optimierung für libFirm wurde in [Hel12] vorgestellt.

Die Eliminierung partieller Redundanzen lässt sich nicht nur für arithmetische Berechnungen durchführen, sondern auch für lesende Speicherzugriffe. Abbildung 6.3 zeigt eine Konstellation mit zwei Load-Knoten, die auf diese Weise optimiert werden können, in der alten und in der neuen Modellierung. Auf dem rechten Ausführungspfad wird zweimal von der Adresse gelesen. Zur Eliminierung dieser Redundanz kann der untere Load-Knoten in den linken Grundblock verschoben werden. Da im unteren Block der geladene Wert aber noch benutzt wird, erhöht sich wieder der Registerdruck.



(a) alte Modellierung (b) neue Modellierung

Abbildung 6.3: Der untere Load-Knoten ist partiell redundant. Durch die neue Modellierung benutzen beide Load-Knoten denselben Speicher und können leicht als gleich erkannt werden.

Durch die neue Modellierung von Lesezugriffen lässt sich auch diese optimierbare Situation leichter erkennen. Da Load-Knoten keinen Speicher mehr produzieren, entfällt der

Phi M-Knoten und die Load-Knoten, die den gleichen Wert liefern, benutzen denselben Speicherwert, sodass sie durch einen einfachen Vergleich von Knotentyp und Eingängen als gleich erkannt werden können.

6.5 Schleifeninvariantes Laden eines Wertes

Wird in einer Schleife in jedem Schleifendurchlauf von einer schleifeninvarianten Adresse geladen, ohne dass der Wert an dieser Speicherstelle zwischenzeitlich überschrieben wird, so lässt sich auch dieser Lesezugriff mithilfe der Alias-Information im FIRM-Graphen und der neuen Modellierung von Load-Knoten leicht erkennen und optimieren. Durch den Einbau der Alias-Information in den FIRM-Graphen werden alle weiteren Speicherzugriffe, die nicht mit dem Load-Knoten in der Schleife aliasen, parallelisiert. Damit entsteht zunächst eine eigene Schleife für den Speicherwert, der vom Load-Knoten benutzt wird. Da der Load-Knoten aber keinen neuen Speicherwert erzeugt, benutzt er in jedem Schleifendurchlauf den vom Beginn der Schleife und es ist kein Phi M-Knoten mehr für diesen Speicherwert nötig. Dadurch kann der Load-Knoten leicht als schleifeninvariant erkannt werden, weil alle seine Operanden vor der Schleife platziert, also schleifeninvariant sind, und der Load-Knoten kann ebenfalls vor die Schleife verschoben werden. Der Lesezugriff wird jetzt nur noch einmal vor der Schleife anstatt in jedem Schleifendurchlauf durchgeführt. Gleichzeitig braucht der geladene Wert aber ein zusätzliches Register während der gesamten Ausführung der Schleife. Für diese Optimierung ist nicht einmal eine neue Regel in Form eines Graph-Musters nötig; nachdem die Speicherzugriffe parallelisiert und in die neue Modellierung überführt wurden, verschiebt die Code-Platzierung den Load-Knoten automatisch vor die Schleife.

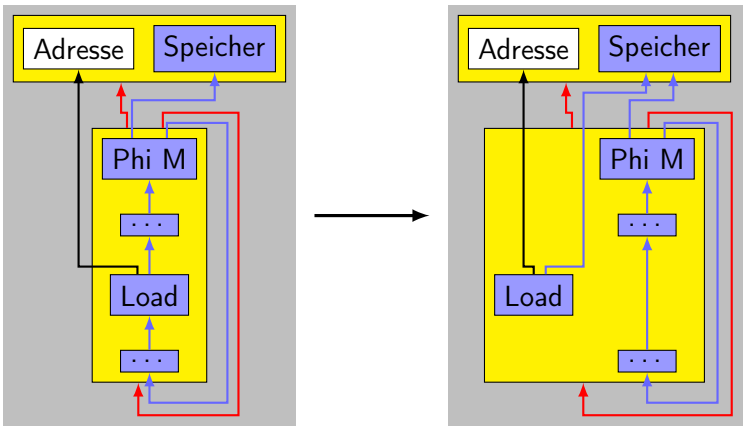


Abbildung 6.4: Nachdem die anderen Speicherzugriffe parallelisiert und der Speicherausgang entfernt wurden, ist der Load-Knoten offensichtlich schleifeninvariant. Jetzt kann er von der Code-Platzierung vor die Schleife geschoben werden.

6.6 Überschreiben eines Wertes in Schleifen

Zwei völlig neue optimierbare Muster wurden für Schleifen implementiert. Das erste ist die Erweiterung der in Abschnitt 6.2 beschriebenen Optimierung von mehrfachen Schreibzugriffen auf Schleifen. Wird in einer Schleife in jedem Durchlauf an eine schleifeninvariante Adresse geschrieben, reicht es nur den letzten Wert einmal nach der Schleife in den Speicher zu schreiben. Innerhalb der Schleife werden die geschriebenen Werte zu einem normalen SSA-Wert, der in einem Prozessorregister verbleiben kann. Abbildung 6.5 zeigt diese Optimierregel im FIRM-Graphen. Der Store-Knoten muss in einen eigenen neuen Grundblock, falls der Grundblock nach der Schleife auch über einen anderen Pfad erreicht werden kann. Durch die Parallelisierung mittels Alias-Information ist diese Situation ein lokales Muster aus einem Phi M- und einem Store-Knoten.

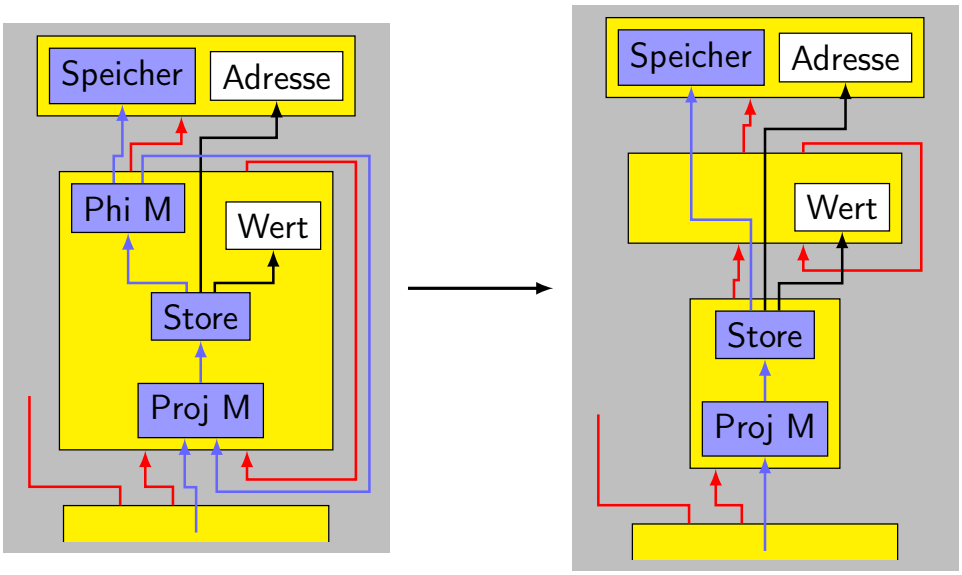


Abbildung 6.5: Der Store-Knoten überschreibt in jedem Durchlauf der Schleife den Wert des letzten Durchlaufs. Deshalb reicht es den Wert nur einmal nach der Schleife zu speichern.

Allerdings gibt es zwei Einschränkungen bei dieser Optimierung zu beachten. Die Schleife darf nur einen Austrittspunkt besitzen. Sonst können die Benutzer des Proj M-Knotens und die Steuerflusskanten, die die Schleife verlassen, einander nicht zugeordnet werden. Außerdem muss der ursprüngliche Grundblock des Store-Knotens den Grundblock des Phi M-Knotens postdominieren. In der Abbildung sind die Grundblöcke sogar gleich, das ist aber nicht zwingend erforderlich. Durch die Postdominanz ist sichergestellt, dass der Schleifenrumpf und damit auch der Store-Knoten bereits ohne diese Optimierung mindestens einmal ausgeführt worden wäre. Nach Anwendung der Optimierung wird der Store-Knoten nämlich auch ohne Betreten der Schleife ausgeführt, was im Falle eines Speicherzugriffsfehlers zu einer Änderung des Programmverhaltens führen würde.

Schleifen dieser Art lassen sich auch optimieren, wenn ein zusätzlicher Load-Knoten im Schleifenrumpf vorkommt, der auch von derselben Adresse liest. Steht dieser Load-Knoten hinter dem Store-Knoten, so kann erst der Load-Knoten durch die Optimierung aus Abschnitt 6.1 eliminiert und dann diese Optimierung durchgeführt werden. Dazu müssen diese beiden Optimierregeln in der richtigen Reihenfolge angewendet werden. Steht der Load-Knoten vor dem Store-Knoten, lässt sich dies auch optimieren. Dazu ist allerdings ein etwas erweitertes Muster nötig, das in Abbildung 6.6 dargestellt ist. Hier wird zusätzlich der Load-Knoten vor die Schleife geschoben. Für den geladenen Wert entsteht ein Phi-Knoten.

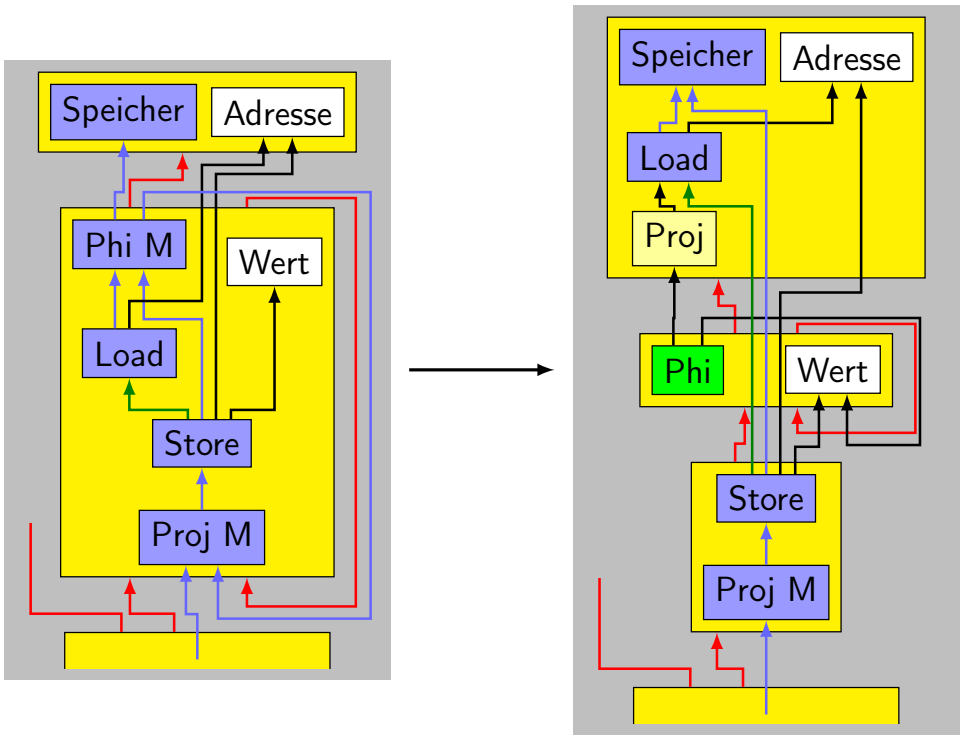


Abbildung 6.6: Der Load-Knoten kann vor, der Store-Knoten hinter die Schleife verschoben werden.

Darüber hinaus ließen sich auch Schleifen, deren Rumpf nicht zwangsläufig einmal ausgeführt wird optimieren. Dazu müsste aber die Schleifenbedingung dupliziert und einmal extra vor der Schleife geprüft werden. Load- und Store-Knoten, die aus der Schleife heraus gezogen wurden, dürfen dann nur ausgeführt werden, wenn die Schleife mindestens einmal ausgeführt wird. Damit wäre sichergestellt, dass keine potentiellen Speicherzugriffsfehler in das Programm eingefügt werden.

In einigen Fällen könnte auch auf die duplizierte Schleifenbedingung verzichtet werden, wenn garantiert werden kann, dass der Speichzugriff über die Adresse nie zu einem Fehler führt. Das ist etwa bei globalen und lokalen Variablen immer der Fall. Wenn die Schleife

allerdings nie betreten wird, wird der Wert an der Adresse dann trotzdem geladen und wieder zurückgeschrieben. Das Programm wird dadurch langsamer.

Alle in diesem Abschnitt beschriebenen Schleifenoptimierungen profitieren wieder davon, dass durch die Repräsentation der Alias-Information im Firm-Graph weitere Speicherzugriffe in der Schleife, die aber nicht mit den Knoten im Muster aliasen, parallelisiert wurden.

7 Evaluation

Alle vorgestellten Verfahren und Optimierungen wurden im Rahmen dieser Arbeit in der Bibliothek libFirm implementiert, um sie so weiter untersuchen zu können. Die angestrebten Vorteile, wie sie in Abschnitt 3.4 formuliert wurden, lassen sich in den kompilierten Programmen wiederfinden. Eine Analyse des vom Compiler produzierten Maschinencodes zeigt, dass die gewonnenen Freiheiten beim Scheduling den Registerdruck verringern, was wiederum zu weniger Spill-Code führt. Dadurch wird in einigen Fällen sogar der Activation Record einer Funktion kleiner, das heißt, auch der Speicherbedarf des Programms lässt sich reduzieren. Die Optimierung der Speicherzugriffe zeigt ähnlich gute Ergebnisse wie ohne die Repräsentation der Alias-Information im Firm-Graph. Allerdings werden diese Optimierungen nun als lokale Muster durchgeführt und nicht, indem die Alias-Information explizit abgefragt und berücksichtigt wird. Zusätzlich kommen die neuen Optimierungen für Speicherzugriffe in Schleifen zum Einsatz. Auch die Vorteile für die Instruktionsauswahl lassen sich im erzeugten Maschinencode beobachten. Mehrere Befehle können dort zu einem zusammengefasst werden, indem der Adressierungsmodus für Speicheradressen als Operanden arithmetischer Operationen ausgenutzt wird. Für eine Beurteilung der Compiler-Performanz ist allerdings zu beachten, dass die Instruktionsauswahl aktuell beide Modellierungen von Lesezugriffen, mit und ohne Speicherausgang, unterstützt um ein Abschalten der neuen Optimierung zu ermöglichen.

Auch wenn die Vorteile in einigen Fällen sichtbar werden, bleibt der tatsächliche Nutzen der neuen Optimierung in der Praxis zu klären. Dazu wurden einige Messungen durchgeführt. Als Testrechner kam ein normaler Arbeitsplatzrechner mit einem Intel Pentium Dual Core E5300 (2,6 GHz), 3.8 GiB Hauptspeicher und Ubuntu 10.04 zum Einsatz.

7.1 Programmlaufzeiten

Das wichtigste Kriterium zur Beurteilung der neuen Optimierung ist die Laufzeit des erzeugten Maschinencodes. Schließlich ist es gerade das Ziel jeder Optimierung diese zu verkürzen.

Zum Projekt libFirm gehört eine Testsuite mit über 1000 Testfällen. Die meisten davon sind sehr klein und testen nur eine spezielle Funktionalität von libFirm oder wurden bei der Behebung eines bestimmten Fehlers erstellt um auf diesen zu testen. Ein paar im-

plementieren aber auch Lösungen zu klassischen Problemen, zum Beispiel verschiedene Sortieralgorithmen, die Türme von Hanoi oder den Verschlüsselungsalgorithmus Serpent und sind damit etwas praxisnäher. Diese 17 Testfälle wurden mit der neuen Optimierung jeweils mit statischer und flusssensitiver Partitionierung kompiliert und zum Vergleich auch mit dem unmodifizierten libFirm ohne Repräsentation der Alias-Information im Firm-Graph. Mithilfe des Werkzeugs Valgrind wurde für jeden Testfall die Anzahl der ausgeführten Maschinenbefehle gemessen. Das Ergebnis ist in Tabelle 7.1 zusammengestellt. Die gemessenen Instruktionen sind jeweils auf Tausend gerundet.

Testfall:	unmodifiziert	stat. Part.	Veränderung [%]	flusss. Part.	Veränderung [%]
QuickSort.c	20.122	21.148	5,10	21.148	5,10
fib.c	26.383	26.374	-0,03	26.374	-0,03
fasta.c	6.105	5.942	-2,66	5.968	-2,24
fannkuch.c	93.048	93.048	0,00	93.048	0,00
spectral-norm.c	93.954	93.965	0,01	93.965	0,01
vpr0.c	108.173	108.173		108.173	
SieveBits.c	616	606	-1,62	606	-1,62
crafty.c	86.444	86.444	0,00	86.444	0,00
MergeSort.c	93.898	94.303	0,43	94.303	0,43
Sieve.c	191.737	191.737		191.737	
Hanoi.c	28.421	28.333	-0,31	28.333	-0,31
bintree.c	129.451	130.222	0,60	130.222	0,60
partial-sums.c	17.286	17.286		17.286	
n-body.c	3.163	3.165	0,06	3.165	0,06
HeapSort.c	38.833	35.719	-8,02	35.719	-8,02
queens.c	118.223	118.223		118.223	
serpent.c	133.038	128.161	-3,67	128.084	-3,72
Durchschnitt:			-0,56		-0,54

Tabelle 7.1: Ein Auszug aus der Firm-Testsuite. Hier wurden die ausgeführten Maschinenbefehle bei Verwendung verschiedener Konfigurationen von libFirm mit dem Werkzeug Valgrind gemessen und der prozentuale Unterschied berechnet. Messwerte in Tausend.

In vier der 17 Testfälle zeigt die neue Optimierung gar keinen Effekt, in weiteren fünf liegt die Änderung unter 0,1 %. Insgesamt lässt sich aber noch eine durchschnittliche Verringerung der Programmlaufzeiten von etwas über 0,5 % durch die neue Optimerung feststellen. Besonders positiv fällt der Testfall HeapSort.c mit einer Verbesserung von 8,02 % auf. Hier wirken sich besonders die neuen Muster für Speicheroperationen in Schleifen aus. Die Tests zeigen aber auch, dass die Laufzeit mit der neuen Optimierung länger werden kann. Trauriger Spitzenreiter ist der Testfall QuickSort.c mit einer Verschlechterung von 5,1 %. Die Verschlechterung kommt von der in Abschnitt 6.3 beschriebenen Optimie-

rung zur Eliminierung wiederholter Lesezugriffe. Im Fall des QuickSort erhöht sich der Registerdruck dadurch so stark, dass ein einmal geladener Wert in den Activation Record ausgelagert und vor seiner letzten Verwendung wieder geladen werden muss. Eigentlich könnte das Auslagern entfallen und von der ursprünglichen Adresse noch einmal geladen werden. Dass sich der Wert an dieser Adresse nicht verändert hat, wird vom Registerallokator jedoch nicht erkannt. Diese Situation wiederholt sich im QuickSort durch das mehrmalige Inlining eines rekursiven Funktionsaufrufs mehrfach und führt dadurch zur beobachteten Verschlechterung. Schaltet man die Eliminierung wiederholter Lesezugriffe ab, wird der QuickSort deutlich schneller, insbesondere auch schneller als mit dem unmodifizierten libFirm. Der gleiche Effekt, allerdings in schwächerer Form, tritt auch bei MergeSort.c und bintree.c auf, andere Testfälle profitieren dagegen durchaus von der Eliminierung wiederholter Lesezugriffe. Diese Optimierung ist also trotz der möglichen Verschlechterung sinnvoll.

Der Vergleich zwischen statischer und flusssensitiver Partitionierung fällt bei diesen Testfällen recht ernüchternd aus. Obwohl sich leicht kleine Testfälle konstruieren lassen, in denen die statische Partitionierung im Gegensatz zur flusssensitiven nichts ausrichtet, zeigt sich in der Testsuite kein Vorteil des mächtigeren Partitionierungsverfahrens. Die meisten Parallelisierungseffekte in den Testfällen entstehen durch die Abspaltung globaler Variablen, auf die keine weiteren Zeiger erzeugt werden, vom restlichen Speicher. Das leistet aber auch die statische Partitionierung.

Testfall:	unmodifiziert	stat. Part.	Verbesserung [%]	flusss. Part.	Verbesserung [%]
164.gzip	1397	1402	0,36	1399	0,14
175.vpr	1511	1547	2,38	1542	2,05
176.gcc	2437	2437	0,00	2475	1,56
181.mcf	1781	1826	2,53	1847	3,71
186.crafty	2012	2056	2,19	2061	2,44
197.parser	1583	1588	0,32	1590	0,44
253.perlbmk	2086	2141	2,64	2137	2,44
254.gap	2088	2023	-3,11	2020	-3,26
255.vortex	1985	1961	-1,21	1946	-1,96
256.bzip2	1622	1626	0,25	1648	1,60
300.twolf	2702	2725	0,85	2793	3,37
Durchschnitt:	1928	1939	0,60	1951	1,20

Tabelle 7.2: Die Messergebnisse der SPECint2000. Angegeben sind die Verhältnisse der gemessenen Laufzeiten zu den SPEC Referenzzeiten und die prozentualen Unterschiede zum unmodifizierten libFirm.

Um die Testbasis zu erweitern wurde auch mit der SPECint2000 gemessen. Dabei kamen die gleichen Versionen von libFirm zum Einsatz wie bei der Testsuite. Tabelle 7.2 zeigt

die Verhältnisse der gemessenen Laufzeiten zu den SPEC Referenzzeiten und die prozentualen Unterschiede zur Übersetzung mit dem unmodifizierten libFirm. Der Testfall 252.eon ist nicht enthalten, weil das Frontend „cparser“, das zum Aufbau der initialen Firm-Graphen verwendet wurde, die Quellsprache C++ nicht beherrscht.

Bei diesen Testfällen zeigt sich die Stärke der flusssensitiven Partitionierung. Der durchschnittliche Geschwindigkeitsgewinn ist doppelt so groß wie mit statischer Partitionierung. Der Effekt der flusssensitiven Partitionierung ist fast immer stärker, als wenn mit statischer Partitionierung kompiliert wurde. Dies gilt allerdings auch für die Programme 254.gap und 255.vortex, bei denen der Effekt negativ ist. Bei C-Programmen dieser Größe kann die flusssensitive Partitionierung also ihr Potential entfalten.

7.2 Compilezeiten

Neben ihrem Nutzen in Form von schnelleren Programmlaufzeiten verursacht die neue Optimierung natürlich auch mehr Aufwand beim Kompilieren. Um die neue Optimierung beurteilen zu können, wird daher in diesem Abschnitt auch ihr Einfluss auf die Laufzeit des Compilers untersucht. Dazu wurden die Laufzeiten der einzelnen Phasen der neuen Optimierung zunächst grob im O-Kalkül abgeschätzt.

In Phase I werden alle Speicherknoten einmal mittels Breitensuche besucht und gesammelt. Dabei bezeichne n die Anzahl der Load- und Store-Knoten im Firm-Graph, $\tilde{n}$ die Anzahl aller Speicherzugriffe einschließlich Call- und Return-Knoten und $\hat{n}$ die Anzahl aller Speicherknoten, also noch einschließlich der Hilfsknoten Proj M, Phi M und Sync. Weiter sei m die Anzahl aller Speicher- und Abhängigkeitskanten. Dann ist die Breitensuche in $O(\hat{n} + m)$ möglich. Zusätzlich wird in Phase I eine Instruktionsliste in jedem Grundblock angelegt. Da jeder Knoten in genau einem Grundblock liegt und nur zu dessen Liste hinzugefügt wird, ginge das in $O(\tilde{n})$. Allerdings müssen die Knoten in den Listen noch paarweise auf weitere Datenabhängigkeiten untereinander untersucht werden (vgl. Abschnitt 5.1). Das verursacht quadratischen Aufwand $O(\tilde{n}^2)$. Insgesamt liegt die Laufzeit dieser Phase also in $O(\hat{n} + m + \tilde{n}^2)$. Wegen $m \leq \hat{n}^2$ und $\tilde{n} \leq \hat{n}$ lässt sich das zu $O(\hat{n}^2)$ vereinfachen. Das ist aber nur noch eine sehr grobe Abschätzung, da der Firm-Graph meistens recht dünn und schon aufgrund der Proj M-Knoten $\tilde{n}$ wesentlich kleiner als $\hat{n}$ ist.

Zur Berechnung der Partitionierung in Phase II werden nur die n Load- und Store-Knoten betrachtet, weil nur für diese eine Alias-Analyse möglich ist. Das Verfahren für die statische Partitionierung verursacht dabei quadratischen Aufwand $O(n^2)$, weil jeder Knoten einmal mit allen schon bearbeiteten Knoten auf Aliasing untersucht wird und im Falle einer Verschmelzung von Partitionen jeder Knoten höchstens einmal in eine andere Partition kopiert werden muss. Die statische Partitionierung berechnet die Zusammenhangskomponenten des Alias-Graphen, was mit der Union-Find-Datenstruktur eigentlich

schneller möglich wäre. Allerdings würde dazu die Kantenliste des Graphen benötigt. Der Alias-Graph wird jedoch nie explizit angelegt, sodass die Kantenliste nicht zur Verfügung steht und stattdessen für alle Knotenpaare eine Alias-Abfrage durchgeführt werden muss. Das verursacht in jedem Fall quadratischen Aufwand. Eine Verbesserung der asymptotischen Laufzeit ist so also nicht möglich.

Für die flusssensitive Partitionierung wurde bereits in Abschnitt 5.2.2 gezeigt, dass das Ergebnis und daher auch die Laufzeit exponentiell groß werden können. Eine konkrete Aussage über die Variante der flusssensitiven Partitionierung mit Heuristik lässt sich leider nicht zeigen. Zumindest aber lehnt die Heuristik die Partitionierung der in [MM65] beschriebenen, extremalen Graphenfamilie mit $3n$ Knoten und 3^n maximalen Cliquen ab. Diese Graphen sind das Komplement aus n Dreier-Cliquen, das heißt, die Knoten lassen sich in Dreiergruppen einteilen, sodass jeder Knoten genau mit den Knoten, die nicht in seiner Gruppe sind, durch eine Kante verbunden ist. Das Verfahren würde bei einem solchen Graphen im ersten Schritt die Knotenmenge in $3n$ neue Mengen der Größe $3n - 2$ zerlegen, was von der Heuristik für $n \geq 3$ abgelehnt wird. Diese Eigenschaft des heuristischen Verfahrens lässt sich auch noch genauer beschreiben. Dazu wird das Verfahren in einzelne Runden unterteilt. In jeder Runde wird jede der aktuellen Knotenmengen einmal aufgeteilt und die so erhaltenen Mengen bilden den Ausgangspunkt der nächsten Runde. Knotenmengen, die nicht mehr aufgeteilt werden können, werden aus dem Verfahren entfernt. Ausgehend von der Menge aller Speicherzugriffe als erste aktuelle Menge ergibt das einen Aufteilungsbaum wie er in Abbildung 7.1 angedeutet ist.

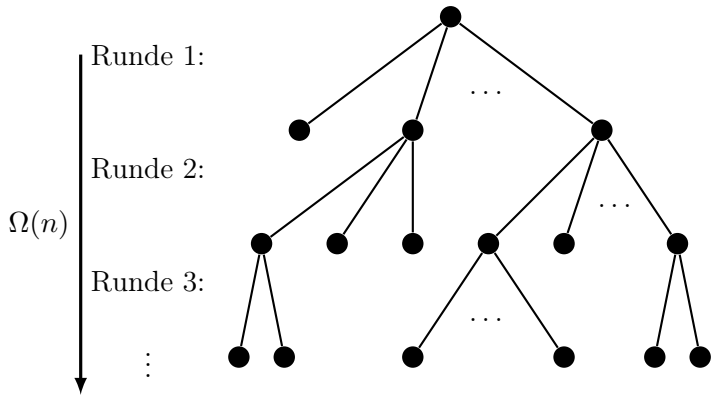


Abbildung 7.1: Ein schematischer Aufteilungsbaum bei der flusssensitiven Partitionierung. In jeder Runde ist eine Knotenmenge (ein Knoten des Baumes) entweder fertig oder sie wird durch mehrere echte Teilmengen (Kindknoten) ersetzt.

Da die Knotenmengen von Runde zu Runde mindestens um eins kleiner werden, kann es nach einer Runde nur noch so viele weitere geben, wie die größte aktuelle Menge Elemente besitzt. Damit eine Familie von Graphen zu einer Unterteilung in $\Omega(c^n)$ mit $c > 1$ vielen Cliquen führt, muss es im zugehörigen Aufteilungsbaum $\Omega(n)$ viele Runden geben.

Dazu müssen auf mindestens einem Pfad von der Wurzel des Aufteilungsbaums zu einem Blatt immer wieder Knotenmengen entstehen, die fast so groß sind wie die Menge, aus der sie hervorgehen. Die Mengen auf einem solchen Pfad brauchen zur Konstruktion immer einen anderen Knoten, mit dem viele aber nicht alle anderen Knoten aliasen. Diese Knoten haben aber auch schon in den Runden vorher eigene, recht große Mengen erzeugt, weil schon Knoten, mit denen sie nicht aliasen, in der gleichen Menge vorhanden waren. Ein sehr hoher Aufteilungsbaum besitzt demnach auch einen hohen Verzweigungsgrad. Wenn in einem Aufteilungsschritt nun zu viele große Knotenmengen entstehen, verletzen diese das Kriterium der Heuristik, dass eine Aufteilung höchstens fünfmal so viele Knoten wie die Ausgangsmenge enthalten darf. Die Heuristik beschränkt also die Kombination aus Höhe und Verzweigungsgrad des Aufteilungsbaums. Gleichzeitig lehnt sie dabei genau die Partitionierungen ab, die eine große Überlappung haben und deshalb nur wenig Parallelisierungspotenzial bieten.

Der Aufwand der Phase III hängt neben der Größe des Firm-Graphen auch von der Struktur des Steuerflussgraphen, insbesondere der Anzahl b an Grundblöcken, und von der Größe p der in Phase II gefundenen Partitionierung ab. Beim ersten Besuch jedes Grundblocks wird die Instruktionsliste einmal durchlaufen und es werden neue Speichereingänge ermittelt. Dazu muss für jede Speicheroperation geprüft werden, zu welchen Speicherpartitionen sie gehört. Das benötigt also $O(\tilde{n} * p)$ Zeit. Zusätzlich wird jeder Grundblock so oft besucht, wie er Vorgänger hat, und beim ersten Besuch werden Phi M-Knoten erzeugt und später vervollständigt. Dabei gibt es in jedem Grundblock zu jeder Speicherpartition ein Phi M und jedes besitzt Vorgänger($Block$) viele Eingänge. Die gesamte Phi M-Konstruktion läuft in $\sum_{i=1}^{b} \text{Vorgänger}(Block_i)^2 * p$ Zeit. Die Anzahl der Vorgänger jedes Grundblockes könnte dabei grob nach b abgeschätzt werden, mehr als vier Vorgänger sind aber bereits extrem selten. Schließlich müssen in dieser Phase noch die Abhängigkeitskanten berechnet werden. Dabei gibt es zu jedem Lesezugriff für jede benutzte Speicherpartition eine Abhängigkeitskante vom nächsten Schreibzugriff auf jedem Ausführungspfad. Zwar wäre der Aufwand für eine Abhängigkeitskante pro Speicherpartition und Lesezugriff bereits durch die Berechnung der Speichereingänge in $O(\tilde{n} * p)$ abgedeckt und oft liegt der nächste Schreibzugriff im selben Grundblock wie der Lesezugriff. Da es aber theoretisch Ausführungspfade geben kann, sodass in fast jedem anderen Grundblock ein möglicher nächster Schreibzugriff auf die Speicherpartition erfolgt, ist der Aufwand für die Abhängigkeitskanten im Allgemeinen nur durch $O(\tilde{n} * p * b)$ begrenzt.

Neben den asymptotischen Laufzeitabschätzungen wurden auch die Compilezeiten der SPECint2000 ermittelt. Als Vergleich zur statischen und flusssensitiven Partitionierung kam hier das modifizierte libFirm allerdings ohne die neue Optimierung zur Repräsentation der Alias-Information zum Einsatz um nur die Laufzeitdifferenz dieser Optimierung zu ermitteln. Die Ergebnisse sind in Tabelle 7.3 zusammengestellt.

Das Verfahren zur statischen Partitionierung verlängert die Compilezeit der gesamten SPECint2000 kaum. In mehreren Testfällen verkürzt sich die Compilezeit sogar. Der

Testfall:	modifiziert, ohne Part.	stat. Part.	Verlängerung [%]	flusss. Part.	Verlängerung [%]
164.gzip	00:09	00:09	0,00	00:10	11,11
175.vpr	00:27	00:28	3,70	00:29	7,41
176.gcc	07:43	07:40	-0,65	07:50	1,51
181.mcf	00:03	00:03	0,00	00:03	0,00
186.crafty	00:51	00:54	5,88	00:56	9,80
197.parser	00:37	00:38	2,70	00:39	5,41
253.perlbmk	03:29	03:29	0,00	03:34	2,39
254.gap	02:18	02:16	-1,45	02:23	3,62
255.vortex	02:00	02:01	0,83	02:05	4,17
256.bzip2	00:13	00:14	7,69	00:15	15,35
300.twolf	00:46	00:45	-2,17	00:55	19,57
Gesamt:	18:36	18:37	0,09	19:19	3,85

Tabelle 7.3: Compilezeiten der SPECint2000 in [mm:ss]. Hier wurden nur verschiedene Konfigurationen des modifizierten libFirm verglichen.

Grund hierfür dürfte darin zu finden sein, dass durch die Repräsentation der Alias-Information und den dadurch möglich gewordenen Optimierungen die Firm-Graphen verkleinert werden und nachfolgende Optimierungen davon profitieren. Die Compilezeiten mit flusssensitiver Partitionierung zeigen deutliche Schwankungen. Der Aufwand hängt stärker von der konkreten Alias-Relation und der daraus resultierenden Partitionierung als von der Größe der Firm-Graphen ab. Die Verlängerung der gesamten Compilezeit fällt aber auch hier mit 3,85 % recht moderat aus.

In einem Vergleich mit dem unmodifizierten libFirm zeigten sich deutlich größere Unterschiede bei der Compilezeit. Alle Testfälle der SPECint2000 zu übersetzen dauert mit der neuen Optimierung und den Modifikationen an bestehenden Optimierungen mehr als 10 % länger. Ein Grund für diese Steigerung ist die zusätzliche Entfernung der Phi-Zyklen nach der neuen Optimierung. Weiterhin brauchen die neuen Optimierregeln für Speicheroperationen in Schleifen etwas mehr Zeit.

Den Großteil der Steigerung machen aber die Modifikationen zur korrekten Behandlung von Abhängigkeitskanten aus. Das Entfernen toter Abhängigkeitskanten spielt hier eine wichtige Rolle. Auch die modifizierte Code-Platzierung benötigt deutlich mehr Zeit. Bei letzterer wird der Vorteil der SSA-Form erkennbar, die viele Optimierungen einfacher und schneller macht. Da die Abhängigkeitskanten diese Form nicht erfüllen, erhöht ihre Berücksichtigung den Aufwand der Code-Platzierung. Das vorrangige Ziel von Abhängigkeitskanten und den dadurch nötigen Modifikationen war jedoch die genauere Modellierung von Lesezugriffen in Firm. Der Einfluss auf die Compilezeit wurde dabei zunächst vernachlässigt und es sind sicher noch Verbesserungen an der Implementierung möglich.

Eine detailliertere Messung der einzelnen Compilerphasen am Testfall 256.bzip2 bestätigt diese Ergebnisse noch einmal. Im Vergleich zum unmodifizierten libFirm dauert der Compiliervorgang mit statischer Partitionierung 16,67 %, mit flusssensitiver Partitionierung sogar 25 % länger. Dabei macht die neue Optimierung mit statischer Partitionierung nur 4,2 % der Dauer aller Optimierungen aus, mit flusssensitiver Partitionierung 7,15 %. Die Hauptverursacher für die deutlich längere Compilezeit sind die Code-Platzierung und JumpThreading. Beide brauchen durch die Modifikationen zur Berücksichtigung der Abhängigkeitskanten mehr als 40 % länger.

8 Fazit

In dieser Arbeit wurde eine Möglichkeit entwickelt, Alias-Information in Programmgraphen, speziell in Funktionsgraphen der Zwischensprache Firm, unmittelbar in deren Struktur darzustellen um so unnötige Abhängigkeiten zwischen Speicherzugriffen zu eliminieren und die Alias-Information für andere Optimierungen nutzbar zu machen.

Dazu wurde zunächst die Modellierung für Speicher in Firm erweitert. Die neue Modellierung ermöglicht es, eine Aufteilung des Speichers vorzunehmen und unabhängige Speicherzugriffe verschiedene Teile des Speichers benutzen zu lassen. Darüber hinaus werden auch Lesezugriffe so modelliert, dass sie den Speicher nicht verändern und somit unabhängig und in ihrer Reihenfolge veränderlich sind.

Weiter wurde ein Verfahren entwickelt, dass Firm-Graphen in die neue Modellierung überführt und dabei einige unnötige Abhängigkeiten beseitigt, indem es den verwendeten Speicher aufteilt und die verfügbare Alias-Information in den Graphen einbringt. Das Verfahren wurde im Rahmen dieser Arbeit auch in der Bibliothek libFirm implementiert. Zur Aufteilung des Speichers betrachtet das Verfahren nicht direkt die vom Programm benutzten Daten. Stattdessen analysiert es nur die verfügbare Alias-Information zu den im Programm enthaltenen Speicherzugriffen um so verschiedene Speicherbereiche zu identifizieren, die auch tatsächlich zu einer Verminderung der Abhängigkeiten im Firm-Graph führen können. Für die Aufteilung des Speichers wurden zwei unterschiedlich aggressive Varianten vorgestellt.

Aus der Repräsentation der Alias-Information im Programmgraph mittels der neuen Modellierung ergeben sich zwei vorteilhafte Eigenschaften. Speicherzugriffe, die sich nicht beeinflussen, hängen auch im Firm-Graph nicht voneinander ab, wodurch sich mehr Freiheiten ergeben. Gleichzeitig sind die Speicherzugriffe, die auf gemeinsamen Speicher zugreifen, auch im Firm-Graph nah beieinander, sodass eine gewisse Lokalität zwischen ihnen besteht. Über diese beiden Eigenschaften nutzen andere Optimierungen automatisch die Alias-Information ohne sie explizit berechnen und berücksichtigen zu müssen. Dadurch werden andere Optimierungen verbessert, sodass das zu übersetzende Programm stärker oder einfacher optimiert werden kann. Die Verbesserung der Optimierung durch die Repräsentation der Alias-Information im Firm-Graph ließ sich mit der Implementierung in libFirm auch durch Vergleichsmessungen belegen.

8.1 Ausblick

An dieser Stelle sollen noch drei mögliche Fortführungen der vorliegenden Arbeit aufgezeigt werden. So ließen sich etwa noch die Auswirkungen der neuen Optimierungen auf den Registerdruck erforschen. In Kapitel 6 wurden einige Optimierungen vorgestellt, die den Registerdruck erhöhen können, meist indem sie Load-Knoten entfernen. Anstatt einen Wert neu aus dem Speicher zu laden wird er von vorher weiterbenutzt, belegt dazu aber in der Zwischenzeit ein zusätzliches Register. In Kapitel 7 hat sich gezeigt, dass der erhöhte Registerdruck zu deutlichen Verschlechterungen der Laufzeit führen kann. Diesen Effekt könnte man auf verschiedenen Zielplattformen mit unterschiedlich vielen Registern untersuchen. Die Optimierungen, die den Registerdruck erhöhen, könnten dann so erweitert werden, dass die Graph-Transformation nicht immer durchgeführt wird, sondern erst der Registerdruck abgeschätzt und berücksichtigt wird.

Eine weitere Fortführung betrifft die Alias-Analyse. Da das Verfahren für die Aufteilung des Speichers nicht die Daten sondern nur die Alias-Relation betrachtet, steht und fällt es mit der Genauigkeit der Alias-Analyse. Mit einer mächtigeren Analyse könnten noch mehr Abhängigkeiten zwischen den Speicherzugriffen ausgeschlossen und bessere Ergebnisse erzielt werden. Neben zusätzlichen Regeln zur Ableitung von Alias-Information sollte eine verbesserte Alias-Analyse in libFirm auch Anfragen zu CopyB-Knoten bearbeiten können, sodass diese nicht mehr so behandelt werden müssen, als ob sie auf jeglichen Speicher zugreifen können. Das Konzept der Modes steht dem aber bisher entgegen. Auch die Approximation für Call-Knoten lässt sich verbessern, indem diese Knoten nicht auf Speicher zugreifen, dessen Adresse die aufrufende Funktion nie verlässt oder in der aufgerufenen Funktion nicht sichtbar ist. Die Alias-Analyse müsste dazu auch Call-Knoten von Zugriffen auf derart unerreichbaren Speicher trennen. Darüber hinaus wäre auch eine interprozedurale Analyse denkbar, die den tatsächlich von einer aufgerufenen Funktion benutzten Speicher approximiert.

Die dritte Richtung, in die diese Arbeit fortgeführt werden könnte, ist die Modellierung von Speicher. Einige Probleme mit der Speichermodellierung in Firm, etwa die Vermischung mit der Termination, wurden in Abschnitt 2.1.2 bereits erwähnt. Darüber hinaus könnte noch der Speichereingang des Alloc-Knotens entfernt werden. Stattdessen würden diese Knoten dann selbst neuen Speicher erzeugen. Analog könnte dann der Free-Knoten auch keinen Speicherausgang mehr besitzen, sondern den Speicher, der ihn erreicht, vollständig konsumieren. Diese Änderung hätte aber weitreichendere Konsequenzen, als auf den ersten Blick ersichtlich ist. So wäre der Start-Knoten nicht mehr die Quelle für alle Speicherwerte. Außerdem wäre unklar, welchen Speicherwert ein Return-Knoten benutzt, wenn aller Speicher in einer Funktion freigegeben wurde. Es wäre nun interessant zu untersuchen, wie sich Speicher noch genauer modellieren und wie viel Optimierungspotenzial sich damit noch erschließen lässt.

Literaturverzeichnis

[And94] Andersen, Lars O.: *Program Analysis and Specialization for the C Programming Language*, Dissertation, 1994

[BK73] Bron, Coen; Kerbosch, Joep: Algorithm 457: finding all cliques of an undirected graph. In: *Commun. ACM* 16 (1973), September, Nr. 9, 575–577. – DOI 10.1145/362342.362367. – ISSN 0001–0782

[CCL+96] Chow, Fred C.; Chan, Sun; Liu, Shin-Ming; Lo, Raymond; Streich, Mark: Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In: *Proceedings of the 6th International Conference on Compiler Construction*. London, UK : Springer-Verlag, 1996 (CC' 96). – ISBN 3–540–61053–7, 253–267

[CFR+91] Cytron, Ron; Ferrante, Jeanne; Rosen, Barry K.; Wegman, Mark N.; Zadeck, F. K.: Efficiently computing static single assignment form and the control dependence graph. In: *ACM Trans. Program. Lang. Syst.* 13 (1991), Oktober, Nr. 4, 451–490. – DOI 10.1145/115372.115320. – ISSN 0164–0925

[Che00] Cheng, Ben: Compile-Time Memory Disambiguation for C Programs. Champaign, IL, USA : University of Illinois at Urbana-Champaign, 2000. – Forschungsbericht

[CP95] Click, Cliff; Paleczny, Michael: A simple graph-based intermediate representation. In: *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*. New York, NY, USA : ACM, 1995 (IR '95). – ISBN 0–89791–754–5, 35–49

[DKK99] Dulong, Carole; Krishnaiyer, Rakesh; Kulkarni, Dattatraya: An Overview of the Intel IA-64 Compiler. In: *Intel Technology Journal* (1999)

[Hel12] Helmer, Christian: *SSA-basierte Eliminierung partieller Redundanzen*, Karlsruher Institut für Technologie (KIT), Diplomarbeit, 2012

[Lan92] Landi, William: Undecidability of Static Analysis. In: *ACM Letters on Programming Languages and Systems* 1 (1992), S. 323–337

[LBBG05] Lindenmaier, Götz; Beck, Michael; Boesler, Boris; Geiß, Rubino: Firm, an

Intermediate Language for Compiler Research. Version: März 2005. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000003172`. University of Karlsruhe, März 2005 (2005-8). – Forschungsbericht. – 19 S.

[MM65] Moon, J.; Moser, L.: On cliques in graphs. In: *Israel Journal of Mathematics* 3 (1965), S. 23–28. – DOI 10.1007/BF02760024. – ISSN 0021–2172

[Nov07] Novillo, Diego: Memory SSA – A Unified Approach for Sparsely Representing Memory Operations. In: *Proceedings of the GCC Developers' Summit*, 2007, S. 97–110

[Ste95] Steensgaard, Bjarne: Sparse functional stores for imperative programs. In: *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*. New York, NY, USA : ACM, 1995 (IR '95). – ISBN 0–89791–754–5, 62–70

[Ste96] Steensgaard, Bjarne: Points-to analysis in almost linear time. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1996 (POPL '96). – ISBN 0–89791–769–3, 32–41

Danksagung

Zum Schluss möchte ich mich noch bei allen bedanken, die mich bei der Erstellung der vorliegenden Arbeit unterstützt haben. Insbesondere gilt mein Dank Matthias Braun für die Betreuung dieser Arbeit und für seine Hilfe, wann immer Abhängigkeitskanten in libFirm nicht so leicht unterzubringen waren, sowie Sebastian Buchwald für seine tikz-Bibliothek, mit der sich Firm-Graphen schnell und unkompliziert darstellen lassen.

Außerdem danke ich allen Korrekturlesern und natürlich meinen Eltern, die mir dieses Studium ermöglicht und mich in allen Entscheidungen unterstützt haben.