Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Optimierung von Dynamic Dispatch mit Rapid Type Analysis für FIRM

Studienarbeit von

Steffen Knoth

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. Jörg Henkel
Betreuender Mitarbeiter:	Dipl.-Inform. Andreas Zwinkau

Bearbeitungszeit: 29. Januar 2015 – 29. April 2015

Zusammenfassung

Rapid Type Analysis ist eine leistungsfähige, statische Programmanalyse um dynamisch gebundene Methodenaufrufe durch statisch gebundene zu ersetzen, wo dies möglich ist ohne das Programmverhalten zu ändern. In dieser Arbeit wird die Rapid Type Analysis für das Compilerframework Firm implementiert und insbesondere für den X10-Compiler X10i angewandt.

Rapid Type Analysis is a powerful static program analysis for replacing dynamically bound method calls with statically bound ones where it is possible without changing the behavior of the program. In this thesis the Rapid Type Analysis is implemented for the compiler framework Firm and used for the X10 compiler X10i.

Inhaltsverzeichnis

1. Einführung

1.1. Motivation

Software spielt heutzutage eine immer größere Rolle. Viele Softwaresysteme werden immer umfangreicher und komplexer, da immer mehr Funktionalität hinzugefügt wird. Gleichzeitig soll die Software aber auch leistungsfähig und möglichst fehlerfrei sein. Zu diesem Zweck braucht es Werkzeuge und Programmiersprachenkonzepte, die es ermöglichen, die Softwareentwicklung handhabbar zu machen. Eines der bedeutendsten Konzepte dazu ist die Objektorientierung.

Objektorientierte Programmierung ist heutzutage ein weit verbreiteter Standard. Sie erlaubt eine intuitive Modellierung komplexer Systeme als kooperierende Objekte. Jedes Objekt besitzt Eigenschaften und Methoden. Auf diese Weise können Daten mit dem zugehörigen Code gebündelt werden. Implementierungsdetails können gekapselt werden, um sie unabhängig vom restlichen Programmcode ändern zu können. Desweiteren wird die Wiederverwendbarkeit von Code erleichtert und Code kann abstrakter ausgedrückt werden, was deutlich zur Übersichtlichkeit beiträgt.

Ohne solche Konzepte zur Modularisierung und Wiederverwendbarkeit wären große Softwareprojekte deutlich schwieriger zu realisieren und zu warten. Aber Techniken, um Quellcode abstrakter, wartbarer und flexibler zu machen, führen in der Regel eine Indirektion ein, die erhöhte Laufzeitkosten mit sich bringt.

Ein zentrales Konzept der Objektorientierung ist die dynamische Bindung, mit der bei Methodenaufrufen auf Objekten erst zur Laufzeit entschieden wird, welche Methode tatsächlich aufgerufen wird. Dies passiert innerhalb einer Klassenhierarchie, in der Unterklassen Methoden von ihren Oberklassen entweder erben oder diese überschreiben. Eine Referenz vom statischen Typ einer Oberklasse kann dabei sowohl auf ein entsprechendes Oberklassenobjekt alsauch auf ein Objekt einer Unterklasse verweisen. Wird über diese Referenz eine dynamisch gebundene Methode aufgerufen, so muss entschieden werden, ob die Methode der Oberklasse oder eine Methode einer Unterklasse aufgerufen wird.

Die dynamische Bindung ermöglicht abstrakteren und flexibleren Code, insbesondere durch das Prinzip der Polymorphie, bei dem der Code über die Schnittstelle der Oberklasse auch mit Objekten verschiedener Unterklassen umgehen kann, ohne dass es eine Rolle spielt, wie die konkreten Objekte ausgeprägt sind.

Das Nachschlagen der aufzurufenden Methode zur Laufzeit, was im englischen Sprachraum auch *Dynamic Dispatch* genannt wird, verursacht aber zusätzliche Laufzeitkosten. Zudem stellt die Indirektion eine Barriere für weitere Analysen und Optimierung dar, da zur Übersetzungszeit unbekannt ist, welche Methode aus einer Menge von Methoden an der Stelle aufgerufen wird.

Die dynamische Bindung wird gerade bei modernen Programmiersprachen sehr häufig verwendet, was sie zu einem lohnenden Ziel für Optimierungsversuche macht. Die Indirektion durch die dynamische Bindung ist nicht in allen Fällen notwendig und könnte an vielen Stellen eliminiert werden, sofern man sich sicher sein kann, dass im jeweiligen Programm an diesen Stellen in jedem Fall nur genau eine Methode in Frage kommt aufgerufen zu werden.

Ziel ist es, dynamisch gebundene Aufrufe durch statisch gebundene zu ersetzen, was auch als *Devirtualisierung* bezeichnet wird. Neben der Einsparung der Befehle und Speicherzugriffe für das Nachschlagen der aufzurufenden Methode, besteht zudem ein besonderer Vorteil darin, dass statisch gebundenen Methodenaufrufe gegebenenfalls durch weitere Optimierungen wie z.B. Inlining noch deutlich effizienter umgesetzt werden können, was mit dynamisch gebundenen Methodenaufrufen nicht möglich ist.

Ein Ansatz, um festzustellen, an welchen Stellen die dynamische Bindung nicht nötig ist, ist eine statische Programmanalyse, welche zur Kompilierzeit den Code des gesamten Programms analysiert. Statisch bedeutet hierbei unter anderem, dass die Analyse nicht weiß, welche genauen Ablaufpfade das Programm zur Laufzeit einschlagen wird. Vielmehr muss mit allen möglichen Pfaden gerechnet werden, was solche Analysen sehr aufwendig machen kann. Zudem stehen Informationen, die erst zur Laufzeit vorhanden, sind nicht zur Verfügung.

Ausführliche statische Analysen können sehr aufwendig werden, besonders wenn die Ergebnisse sehr präzise sein sollen. Um Kompilierzeiten nicht deutlich zu verlängern, versucht man meist mit einfacheren, aber dafür ungenaueren Analysen zu arbeiten und mit ihnen dennoch einen möglichst großen Effekt zu erzielen.

Eine verbreitete, statische Programmanalyse zur Optimierung von dynamisch gebundenen Methodenaufrufen ist die Rapid Type Analysis [1][2]. Sie zählt zu den einfacheren und damit schnelleren Analysen, kann aber dennoch einen großen Teil der nicht benötigten, dynamisch gebundenen Aufrufe eliminieren.

1.2. Ziel der Arbeit

In dieser Arbeit soll für das Compilerframework Firm, das SSA-Graphen als Zwischendarstellung nutzt, eine Optimierung von Dynamic Dispatch, d.h. von dynamisch gebundenen Methodenaufrufen, implementiert werden. Dazu bietet sich die Rapid Type Analysis (RTA) von Bacon an, die anhand der Klassenhierarchie und der tatsächlich aufgerufenen Funktionen und den dort verwendeten Klassen bestimmt, welche dynamisch gebundenen Aufrufe durch statisch gebundene ersetzt werden können ohne das Programmverhalten zu verändern.

Ziel ist, die RTA in die bestehenden Bibliotheken libFirm und liboo einzufügen und in damit entwickelten Compilerprojekten, insbesondere dem invasiven X10-Compiler X10i, zum Einsatz zu bringen. Im Idealfall sollen mithilfe der RTA auch lange Kompilierzeiten verringert werden, indem ungenutzte Klassen und Methoden nicht vom Backend bei der Codeerzeugung behandelt werden müssen.

1.3. Aufbau der Arbeit

In Kapitel 2 werden zunächst die Grundlagen und die verwendeten Softwareprojekte vorgestellt und verwandte Arbeiten diskutiert. In Kapitel 3 werden Entwurfsentscheidungen und Anpassungen besprochen und die Implementierung erklärt. Danach wird in Kapitel 4 die Implementierung evaluiert. Den Abschluss bildet Kapitel 5 mit einer Zusammenfassung und einem Ausblick auf zukünftige Arbeiten.

2. Grundlagen und verwandte Arbeiten

In diesem Kapitel werden zunächst die theoretischen Grundlagen besprochen und die für die Implementierung verwendeten Projekte vorgestellt. Danach werden verwandte Arbeiten diskutiert.

2.1. Programmanalyse

Programmanalyse ist das automatische Untersuchen von Computerprogrammen. Programmanalysen werden oft für Optimierungen eingesetzt, um vor dem Optimierungsschritt die dafür benötigten Informationen zu gewinnen. Ein weiteres Anwendungsgebiet ist das Finden von typischen Programmierfehlern. Darüberhinaus werden Programmanalysen auch eingesetzt, um das Programm besser verstehen zu können, etwa um Sicherheitsüberprüfungen durchzuführen.

Die statische Programmanalyse untersucht das Programm ohne es auszuführen, beschränkt sich also auf den statischen Programmcode. Dabei müssen alle möglichen Ablaufpfade des Codes berücksichtigt werden, wodurch statische Programmanalysen sehr aufwendig werden können. Zudem stehen Informationen, die erst zur Laufzeit vorhanden sind, nicht zur Verfügung.

Programmanalysen müssen dem Prinzip der konservativen Approximation folgen. Das bedeutet im Zweifel Genauigkeit zu opfern, um auf der sicheren Seite zu sein und das Programmverhalten nicht zu ändern.

Es ist möglich, das komplette Programm an einem Stück zu analysieren, eine sogenannte *Whole Program Analysis*, anstatt einzelne Programmteile oder Übersetzungseinheiten getrennt zu betrachten. Auf diese Weise sind zusätzliche Annahmen und Schlussfolgerungen möglich, die sich ausschließlich auf das jeweilige Programm beziehen. Unter anderem lässt sich feststellen, dass nur die im jeweiligen Programm vorkommenden Codeelemente in Betracht gezogen werden müssen und auch nur auf die Weise, wie sie dort verwendet wurden. Man geht dabei davon aus, dass das Programm eine abgeschlossene Einheit darstellt (*Closed World Assumption*) und dass alles, was nicht vorkommt, für die Belange dieses Programms nicht existiert und damit keine Rolle spielt. Das bringt einige Vorteile mit sich, aber auch die Schwierigkeit dafür zu sorgen, dass die Annahme in jedem Fall ihre Gültigkeit behält.

Programmanalysen werden häufig danach kategorisiert ob sie die Reihenfolge der Operationen beachten oder nicht (Flusssensitivität) und ob sie den Kontext von Methodenaufrufen in Betracht ziehen (Kontextsensitivität). Im Allgemeinen gelten flusssensitive und kontextsensitive statische Analysen als deutlich aufwendiger, liefern aber auch genauere Ergebnisse.

2.2. Rapid Type Analysis

Die Rapid Type Analysis (RTA) ist eine einfache und dadurch performante statische Programmanalyse zur Optimierung von dynamisch gebundenen Aufrufen. Die RTA ist weder fluss- noch kontextsensitiv und damit sehr schnell. Da sie trotzdem sehr leistungsfähig ist, hat sie weite Verbreitung gefunden. Die Basis bilden die Arbeiten von Bacon und Sweeney [1][2]. In ihren Arbeiten befassten sie sich hauptsächlich mit C++, die Funktionsweise der RTA ist aber auch auf andere objektorientierte Programmiersprachen übertragbar.

Der RTA liegt das ältere Verfahren der Class Hierarchy Analysis (CHA) [3] zugrunde. Die Grundidee der CHA ist, sich die Informationen in der Klassenhierarchie zunutze zu machen. Dabei wird vorausgesetzt, dass die Klassenhierarchie vollständig gegeben ist, d.h. dass alle Klassen bekannt und eingetragen sind, und dass keine Änderungen daran stattfinden. Die CHA stellt also bereits eine Closed-World-Assumption an.

Für die CHA ist ein dynamisch gebundener Aufruf devirtualisierbar, wenn in der Klassenhierachie unterhalb des statischen Typs der Referenz des Aufrufs keine überschreibenden Definitionen der aufgerufenen Methode zu finden sind. Dies ist insbesondere bei Methoden von Klassen der Fall, die keine Unterklassen aufweisen.

Die RTA erweitert das Vorgehen der CHA um Beobachtung der Objekterzeugung und um eine weitere Closed-World-Assumption, dass keine anderen Objekte vorhanden sind außer denjenigen deren Erzeugung im analysierten Code gefunden werden kann. Gegebenfalls initial vorhandene Objekte müssen der RTA bekannt gemacht werden. Die Idee dabei ist, dass Methoden von Klassen, von denen keine Objekte erzeugt wurden, auch kein Ziel eines dynamisch gebundenen Aufrufs sein können. Dabei werden Subobjekte nicht gezählt, da diese stets Teil eines Unterklassenobjekts mit seinen eigenen Informationen für die dynamische Bindung (z.B. VTable) sind.

Die RTA nutzt einen Callgraphen des Programms für den Durchlauf. Ausgehend von der Mainmethode (oder allgemein von einer Menge an Methoden als Eintrittspunkte) werden nur Methoden untersucht, von denen Aufrufe im analysierten Code gefunden werden. Diese Methoden werden als *lebendig* bezeichnet, da sie tatsächlich aufgerufen werden könnten. Um den Callgraphen möglichst klein zu halten, werden Methoden, die mögliche Ziele von dynamisch gebundene Aufrufen sind, nur dann analysiert, wenn auch deren Klasse *lebendig* ist, d.h. wenn irgendwo im lebendigen Code Objekte der Klasse erstellt werden. Der Vorteil ist nicht nur, dass nur der tatsächlich aufgerufene Code analysiert wird, sondern auch, dass Objekterzeugungen in nicht aufgerufenem Code nicht berücksichtigt werden und damit die Menge der lebendigen Klassen möglichst klein gehalten wird.

Die RTA geht optimistisch vor. Sie startet mit leeren Zielmengen der dynamisch gebundenen Aufrufe, die sie während des Durchlaufs nach und nach füllt, wenn Objekterzeugungen von Klassen gefunden werden. Erst am Ende des Durchlaufs sind die Ergebnisse gültig. Devirtualisierungen und gegebenenfalls weitere Optimierungen können somit erst in einem zweiten Durchlauf erfolgen.

Ergebnis der RTA sind noch weiter reduzierte Mengen von Zielen von dynamisch gebundenen Aufrufen als bei der CHA. Die zusätzliche Information über die tatsächlich

genutzten Klassen macht dies möglich. Dadurch können mehr Aufrufe erkannt werden, bei denen die dynamische Bindung nicht benötigt wird.

Das Beispielprogramm in Listing 2.1 enthält eine einfache Klassenhierarchie aus zwei Klassen, `A` ist die Oberklasse von `B`. Beide dynamisch gebundenen Aufrufe in der Mainmethode kann die RTA devirtualisieren. Der erste Aufruf kann bereits mit den Mitteln der CHA als devirtualisierbar erkannt werden, da `B` keine Unterklassen besitzt. Der zweite Aufruf ist nur deshalb devirtualisierbar, da die RTA ermittelt, dass nirgendwo im gesamten Programm ein Objekt von `A` erstellt wird und damit `A` nicht lebendig ist. `B` ist die einzige lebendige Unterklasse, die die Methode `foo()` überschreibt. Somit ist `B::foo()` die einzige mögliche Zielmethode des Aufrufs.

Aber wäre irgendwo im lebendigen Code des Programms eine zusätzliche Instanziierung der Klasse `A` zu finden, auch wenn sie keinen Einfluss auf den Wert der Variable `a` in Zeile 13 und 14 hätte, könnte die RTA den Aufruf in Zeile 14 nicht mehr devirtualisieren, obwohl in diesem Beispiel eindeutig nur ein Objekt der Klasse `B` Ziel des Aufrufs sein kann.

```
1   class A {
2       public void foo() { System.out.println("A::foo()"); }
3   }
4
5   class B extends A {
6       public void foo() { System.out.println("B::foo()"); }
7   }
8
9   public class Example {
10      public static void main(String[] args) {
11          B b = new B();
12          b.foo(); // CHA can resolve this (no subclass)
13          A a = b;
14          a.foo(); // RTA can resolve this (A not live)
15      }
16  }
```

Listing 2.1: Simples Beispiel in Java um die Fähigkeiten der RTA zu verdeutlichen: Die RTA kann beide Aufrufe in `main` optimieren.

Die RTA stellt einige Anforderungen, auf die sie zwingend angewiesen ist, um korrekt zu arbeiten. Sind diese nicht gegeben, ist es möglich, dass das Verhalten des Programms durch falsche Devirtualisierung verändert wird.

Da die RTA auf der CHA basiert, erfordert sie genauso wie diese, dass das gesamte Programm zur Analyse bereitsteht (*Whole Program Analysis*). In erster Linie geht es dabei darum, das Wissen über alle Klassen und deren vollständige Klassenhierarchien in einer geeigneten Datenstruktur zur Verfügung zu haben. Die Klassenhierachien dürfen sich nicht zur Laufzeit verändern, insbesondere dürfen keine weiteren Klassen zur Laufzeit dazukommen. Folglich ist es nicht möglich Klassen dynamisch nachzuladen.

Die RTA erfordert zusätzlich, dass aller Code, der vom Programm aufgerufen werden könnte, zum Analysieren bereitsteht. Die RTA braucht das vollständige Wissen über die Objekte des Programms. Dazu ist es notwendig, dass sie alle Objekterzeugungen im Programmcode finden kann. Das bedeutet, dass sie alle Eintrittspunkte kennen muss, sofern es mehr gibt als die Mainmethode, und dass sie den Callgraphen vollständig durchlaufen und jede Methode analysieren kann.

Desweiteren setzen CHA und RTA voraus, dass der zu analysierende Code typsicher[1] ist. Genauer gesagt, gehen beide von der Annahme aus, dass bei dynamisch gebundenen Aufrufen der dynamische Typ des Objekts stets gleich dem oder eine Unterklasse des statischen Typs der Referenz ist, auf der der Aufruf stattfindet [4]. Bei unsicheren Downcasts kann es Fälle geben in denen die RTA das Programmverhalten durch die Devirtualisierung verändert. Im Allgemeinen ist es nicht möglich diese Fälle statisch zu erkennen.

Bei hinreichend typsicheren[1] Sprachen (wie z.B. Java), bei denen bei jedem Downcast eine Typprüfung stattfindet und gegebenenfalls eine Exception geworfen wird, tritt der Fall nicht auf, was auch schon in [2] erwähnt wird.

Allerdings ist zu beachten, dass Fälle, in denen Aufrufe auf Nullreferenzen auftreten können, auch zu den Fällen gehören, in denen die RTA das Programmverhalten unter bestimmten Umständen verändert. Statt einer erwarteten Exception (Java) bzw. eines Speicherzugriffsfehlers (C++) wird dann die einzige gefundene Zielmethode mit der Nullreferenz als This-Pointer aufgerufen, was ein anderes Verhalten darstellt als ohne Devirtualisierung und nicht in jedem Fall nachträglich zu einer Exception bzw. einem Speicherzugriffsfehlers führt. Dies könnte durch Einfügen einer zusätzlichen Prüfung vor jedem devirtualisierten Aufruf verhindert werden, würde aber die Optimierungswirkung nur wegen seltenen Ausnahmefällen verringern, die sowieso immer Fehlerzustände darstellen.

Für den praktischen Einsatz ist zudem die von Bacon in [2] beschriebene Erweiterung der RTA wichtig, die es ermöglicht, mit der RTA auch partielle Programme zu optimieren. Häufig verwenden Programme externe Bibliotheken, welche erst nach dem Übersetzen zum Programmcode dazugelinkt werden. In gleicher Weise werden die Bibliotheken separat entwickelt und übersetzt.

Da die RTA eine Whole-Program-Analyse ist, muss sie angepasst werden, um auch unter diesen Bedingungen korrekt arbeiten zu können. Die Grundvorraussetzung dabei ist eine Abgrenzung des Bibliothekscodes vom Programm. Die Analyse muss erkennen können welche Codeteile aus Bibliotheken stammen und welche nicht. Der Bibliothekscode muss separat entwickelt sein, damit man davon ausgehen kann, dass er nicht auf Internas des Hauptprogramms zugreift.

Somit sind zwei Modi der Analyse notwendig, in denen das Vorhandensein unbekannten Codes einbezogen wird: die Übersetzung einer Bibliothek ohne Hauptprogramm und die Übersetzung eines Hauptprogramms ohne Bibliotheken. In beiden Fällen wird das Optimierungspotential der RTA durch die getrennte Übersetzung stark reduziert.

[1] Für Typsicherheit gibt es keine eindeutige Definition. Z.B. gehen die Meinungen auseinander ob Laufzeitfehler aufgrund von Typfehlern erlaubt sein sollen oder nicht.

Im Falle einer Bibliothek werden statt einer Mainmethode alle Methoden, die zur öffentlichen Schnittstelle der Bibliothek zählen[2], initial als lebendig betrachtet und als Wurzeln des Callgraphen genommen. Zudem kann der Code, der die Bibliothek nutzt, alle exportierten nicht-finalen Klassen instanziieren. Folglich müssen diese ebenfalls initial als lebendig angenommen werden. Nicht-finale Klassen können im Hauptprogramm auch als Oberklassen verwendet werden. Somit muss bei diesen von der Existenz von unbekannten Unterklassen ausgegangen werden. Werden Objekte davon an den Bibliothekscode übergeben, können deren Methoden auch von dynamisch gebundenen Aufrufen innerhalb der Bibliothek aufgerufen werden. Daraus folgt, dass Aufrufe auf Referenzen vom Typ einer exportierten nicht-finalen Klasse grundsätzlich nicht devirtualisiert werden dürfen. Einige weitere Anpassungen der Analyse sind notwendig für C++ bezüglich Funktionszeigern und Devirtualisierung von virtueller Vererbung.

Im Falle eines Programms in Abwesenheit von genutzten Bibliotheken ist das Vorgehen ähnlich. Prinzipiell müssen ebenfalls alle Methoden in den öffentlichen Schnittstellen der Bibliotheken initial als lebendig angenommen werden, da die Bibliotheken diese intern selbst aufrufen könnten. Alle von den Bibliotheken exportierten Klassen müssen initial als lebendig angenommen werden, da die Bibliotheken diese jeweils intern nutzen könnten. Aufrufe auf einer Referenz vom Typ einer importierten nicht-finalen Klasse dürfen ebenfalls nicht devirtualisiert werden, da die Bibliotheken intern unbekannte Unterklassen haben können und Objekte davon an das Hauptprogramm zurückgeben können.[3] Wenn Unterklassen von importierten Klassen instanziiert werden, die Methoden der importierten Oberklasse überschreiben, können diese unter Umständen aus dem Bibliothekscode aufgerufen werden und müssen deshalb als lebendig markiert und analysiert werden.

2.3. FIRM

Firm ist der Name sowohl einer Zwischendarstellung für Compiler [5] alsauch des mittlerweile daraus entstandenen Compilerframeworks, welches durch die Bibliotheken libFirm und liboo implementiert wird.

Ursprünglich entstand Firm im Jahr 1996 als Zwischendarstellung des Sather-K-Compilers Fiasco. Daher stammt auch der Name Firm, der für *Fiasco's Intermediate Representation Mesh* steht. Später wurde Firm in eine Open-Source-Bibliothek extrahiert, welche vorallem am Lehrstuhl für Programmierparadigmen am Institut für Programmstrukturen und Datenorganisation am Karlsruher Institut für Technologie weiterentwickelt wird [6].

Firm als Zwischendarstellung unterscheidet sich deutlich von klassischen Zwischendarstellungen, welche meist auf Instruktionslisten mit einer Art von Tripelcode basieren. Firm nutzt dagegen Graphen um den Programmcode darzustellt. In Abbildung 2.1 ist ein Beispiel zu sehen.

[2] Im Fall von Java sind das z.B. alle public-Methoden und alle protected-Methoden von nicht-finalen Klassen.

[3] Ein typisches Beispiel in Java ist die Implementierung des Interfaces `Iterator`.

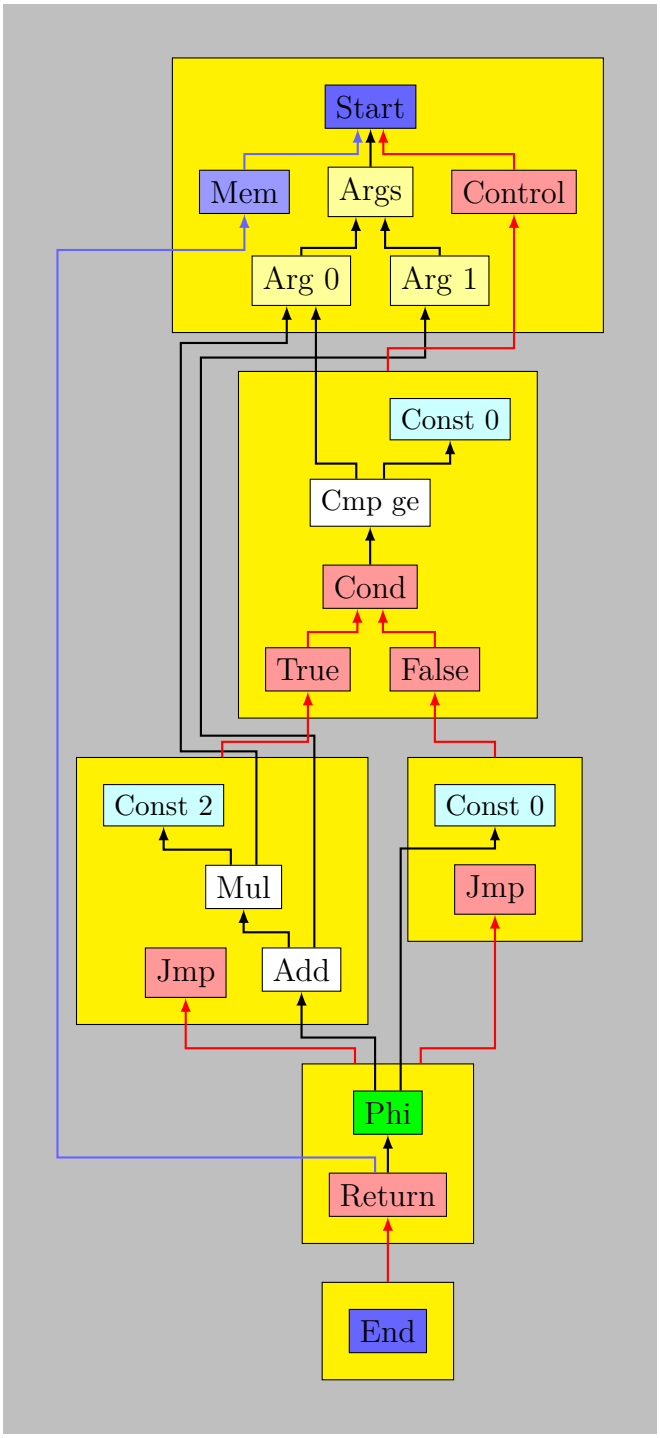


Abbildung 2.1.: Ein FIRM-Graph einer Funktion mit zwei Argumenten x und y, die x auf $>= 0$ prüft und dann $2 * x + y$ oder ansonsten 0 zurückgibt.

Die Graphen in Firm basieren auf Ideen von Click [7] und nutzen die SSA-Form. SSA bedeutet *Static Single Assignment* [8][9] und ist eine Form der Zwischendarstellung von Programmcode, die für Optimierungen und Programmmanalysen besonders vorteilhaft ist. In der SSA-Form wird allen (lokalen) Variablen statisch im Programmtext nur einmal ein Wert zugewiesen. Das hat den Vorteil, dass jede Variable genau eine Definition hat und damit diesem einen Wert entspricht. Damit das funktionieren kann, werden Werte aus getrennten Kontrollflusspfaden mit sogenannten ϕ-Funktionen zusammengeführt, die je nach genommenem Pfad den passenden Wert auswählen. In Graphenform ist es so möglich ganz auf (lokale) Variablen zu verzichten und direkt mit den durch Knoten repräsentierten Werten zu arbeiten. Viele Programme, die sich durch Feinheiten im Programmtext scheinbar unterscheiden, finden sich dadurch in einer annähernd eindeutigen Darstellung wieder.

Die Graphen in Firm sind zudem keine Flussgraphen, sondern Abhängigkeitsgraphen. Das bedeutet vorallem, dass die Verbindungskanten der Graphen in umgekehrter Richtung verlaufen, d.h. vom Endknoten zum Anfangsknoten, und Abhängigkeiten darstellen, wie z.B. Use-Def. Insbesondere verweisen damit Operationen direkt auf ihre Operanden, was sehr nützlich ist, da es häufig vorkommt, dass man sich die Operanden zu einer Operation anschauen will.

Mithilfe der libFirm wird zunächst ein sogenannter Typgraph aufgebaut. In ihm werden alle Typen erstellt. Dabei können komplexere Typen aus primitiven Typen zusammengesetzt werden. Dies wird im Typgraph über Beziehungen zwischen Typknoten dargestellt. Zum Beispiel verweisen Arrays auf ihren Elementtyp und Pointertypen auf den Typ, auf dessen Objekte die jeweiligen Pointer zeigen können.

Neben Typen enthält der Typgraph auch Entities, die als Members in Structs oder Klassen eingefügt werden. Entities sind in Firm eine wichtige Datenstruktur zur Repräsentation sämtlicher Programmbestandteile wie Funktionen, globale Variablen, Felder, usw. Sie tragen Informationen, wie z.B. Name und Typ, und werden zur Referenzierung genutzt. Eine besondere Rolle hat die Klasse GlobalType, die jedes Programm besitzt. Sie enthält per Definition alle globalen Variablen und statischen Funktionen.

Zu Klassen können auch deren Vererbungshierarchien aufgebaut werden. Dazu werden Oberklassen mit ihren Unterklassen bidirektional verbunden. Ebenfalls werden Methoden-Entities mit den zugehörigen überschreibenden und überschriebenen Methoden-Entities verbunden.

Hat man alle benötigten Typen und Entities erstellt, kann man die Programmgraphen aufbauen. Diese enthalten den Code der einzelnen Funktionen als Firm-Graph in SSA-Form. libFirm bietet einen Algorithmus der die Konstruktion der SSA-Graphen inkrementell direkt aus dem abstrakten Syntaxbaum ermöglicht. Dazu ist es nicht erfordlich zuerst einen Kontrollflussgraph oder einen Dominanzbaum aufzubauen [10]. Der Algorithmus setzt auch ϕ-Funktionen selbstständig und macht dadurch den Aufbau des SSA-Graphen sehr einfach.

Firm-Graphen bestehen aus einer Anzahl Firm-Knoten, die untereinander durch Kanten verbunden und auf mehrere Grundblöcke verteilt angeordnet sind. Eine spezielle Rolle spielen der Startblock mit zughörigem Startknoten genauso wie der Endblock mit zugehörigem Endknoten, welche Anfang und Ende darstellen.

Es gibt viele verschiedene Typen von Knoten. Es gibt arithmetische Operationen wie z.B. `Add`, `Sub`, `Mul`, `Div` und `Shift`, Funktionsaufrufe mittels `Call`, und `Load` sowie `Store` für den Zugriff auf nicht-lokale Variablen, Arrays und Felder. Jeder Knoten einer Operation ist wiederum Quelle seiner Ergebniswerte. Für konstante Werte gibt es `Const`-Knoten, `Unknown`-Knoten repräsentieren unintialisierte Werte. `Address`[4]-, `Offset`[4]- und `Member`[5]-Knoten ermöglichen es, Adress- und Offset-Werte symbolisch abzurufen, indem sie auf Entities verweisen. Diese werden insbesondere bei Funktionsaufrufen oder Load/Store-Speicherzugriffen benötigt.

Eine besondere Rolle kommt dem Projektionsknoten (`Proj`) zu, der dafür zuständig ist aus einem Tupel von Werten einen bestimmten Wert zu selektieren. Dadurch wird Komplexität gespart, da auf diese Weise Kanten stets als einfache Zeiger umgesetzt werden können und keine weitere Information tragen müssen. `Proj`-Knoten kommen zum Einsatz wenn ein Knoten mehr als einen Ergebniswert liefert. Mittels `Proj`-Knoten werden beispielsweise auch die Funktionsargumente aus dem Startknoten bezogen.

Wichtig für die SSA-Darstellung sind die `Phi`-Knoten, die die ϕ-Funktionen repräsentieren. Auch sie stehen jeweils für ihren Ergebniswert, den Wert, der je nach genommenem Pfad selektiert wird. Dabei entspricht die Nummer des selektierten Eingangswertes der Nummer des zugehörigen Vorgängerblocks in der Verlinkung des Blocks in dem sich der `Phi`-Knoten befindet.

Operationen und Werte sind in erster Linie über ihre Datenabhängigkeiten verbunden. Dadurch gibt es keine totale Ordnung wie in Befehlslisten, stattdessen wird eine Reihenfolge flexibel über die bestehenden Abhängigkeiten bestimmt. Bei späteren Änderungen am Graphen durch Optimierungen ist es so möglich, dynamisch immer die vorteilhafteste Reihenfolge zu nutzen. Zusätzlich zu Datenabhängigkeiten gibt es eine spezielle, sogenannte Memory-Kante, die benutzt wird, um Abhängigkeiten bei möglichem Aliasing von Speicherzugriffen und bei anderen Seiteneffekten zu modellieren. Üblicherweise gibt die Memory-Kante damit eine Grundreihenfolge im Code innerhalb eines Grundblocks vor, insbesondere von Funktionsaufrufen, Load/Store-Speicherzugriffen und der Return-Anweisung zum Verlassen der aktuellen Funktion. Diese sind häufig die Nutzer der Ergebnisse aus Berechnungen und zeigen somit auch die Reihenfolge der Berechnungsbäume auf. Die endgültige Reihenfolge der Operationen bestimmt erst das Backend bei der Befehlsanordnung im Rahmen der Codegenerierung.

Der Kontrollfluss wird als Kanten zwischen Blöcken und Sprungknoten festgehalten. Bedingte Sprünge und `Return`-Knoten weisen zusätzlich Verbindungen zu Werten auf, die zur Entscheidungsfindung bzw. als Rückgabewert herangezogen werden. Ein Firm-Graph ist damit Kontrollflussgraph und Datenabhängigkeitsgraph in einem.

In Abbildung 2.1 ist ein Beispiel eines Firm-Programmgraphen abgebildet. Er zeigt eine einfache Funktion mit zwei Argumenten x und y, die x auf $>= 0$ prüft und dann $2 * x + y$ oder ansonsten 0 zurückgibt. Die Grundblöcke in Gelb mit zusätzlichem Start- und Endblock enthalten die Knoten. Der Kontrollfluss ist rot dargestellt, Datenabhängigkeiten schwarz. Die Speicherkante in Lila wird hier nicht gebraucht. Der Vergleich

[4] in früheren Versionen `SymConst`
[5] in früheren Versionen `Sel`

mittels `Cmp`-Knoten führt zu einer Verzweigung. Im linken Zweig findet mit `Mul` und `Add` eine Berechnung statt. Der grüne `Phi`-Knoten in dem Block, in dem die Pfade wieder zusammenlaufen, selektiert schließlich entweder den Wert aus dem linken oder aus dem rechten Zweig, welcher dann als Rückgabewert dient.

Durch die geschickte Darstellung des Programmcodes in Firm als SSA-Graph ergeben sich einige inhärente Optimierungen. Da die SSA-Form als Graph ohne lokale Variablen auskommt, gibt es automatisch auch keine Zuweisungen oder Kopien zwischen Variablen (Copy Propagation). Nutzungsstellen von Werten sind direkt mit deren Enstehungsstellen verbunden. Zudem fallen Knoten, die nicht benutzt werden, einfach weg, da diese dann auch nicht verlinkt sind (Dead Code Elimination). Daneben kann libFirm bei der Konstruktion des Firm-Graphen zusätzlich einige Optimierungen on-the-fly durchführen. Direkt beim Anlegen der Knoten wird versucht, Konstanten zusammenzurechnen, gemeinsame Teilausdrücke zu finden und Arithmetik zu vereinfachen. Diese On-the-fly-Optimierungen sind allerdings nur sehr lokal möglich und bei Bedarf auch abschaltbar. Näheres dazu ist in [5] beschrieben.

Für objektorientierte Konzepte gibt es eine zusätzliche Bibliothek namens liboo, die auf die libFirm aufsetzt. Sie bringt neue Knotentypen und Hilfsfunktionen mit um objektorientierte Sprachen bequemer umsetzen zu können. Ein Compiler kann damit zusätzliche Informationen über Knoten und Entities verwalten und die Programmgraphen zuerst mit einigen abstrakten Knotentypen für objektorientierte Konzepte aufbauen. Nach dieser Phase werden diese abstrakten Knoten in einem OO-Lowering durch konkretere Implementierungen ersetzt. Beispielsweise werden dynamische gebundene Methodenaufrufe zunächst mit einem `MethodSel`-Knoten repräsentiert, der dann zum Beispiel durch einen Nachschlagevorgang in Tabellen ersetzt wird. Zudem bietet liboo Unterstützung für die Erstellung solcher Tabellen (VTables) und weiterer Datenstrukturen für Typinformationen zur Laufzeit.

2.4. X10i

X10 ist eine objektorientierte Programmiersprache, die von IBM [11] am Thomas J. Watson Research Center entwickelt wurde. Sie ähnelt Java, enthält aber zusätzlich reichhaltige Erweiterungen für parallele Programmierung. Ziel von X10 ist es insbesondere Programme für heterogene Clustersysteme (*Non-Uniform Cluster Computing*) deutlich effizienter entwickeln zu können. Daher stammt auch der Name X10, weil eine zehnfach höhere Produktivität ermöglicht werden sollte. X10 verwendet das Modell *Partitioned Global Address Space* (PGAS) für die parallele Programmierung, welches es zusätzlich um Asynchronität erweitert: *Asynchronous Partitioned Global Address Space* (APGAS).

X10i [12] ist ein X10-Compiler, der am Lehrstuhl für Programmierparadigmen am Institut für Programmstrukturen und Datenorganisation am Karlsruher Institut für Technologie für das Forschungsprojekt InvasIC [13] entwickelt wird, welches sich mit *Invasive Computing* beschäftigt.

Invasive Computing [14] ist ein neuartiges Programmier- und Verarbeitungsparadig-

ma um zukünftige, heterogene, hochgradig parallele Hardwarearchitekturen besser ausnutzen zu können. Dabei soll ein Programm verfügbare Ressourcen selbstständig und dynamisch erkennen, invadieren und sich genauso auch wieder zurückziehen, wenn die Ressourcen nicht mehr benötigt werden.

X10i wurde auf Basis des X10-Compiler-Frontends von IBM [11] entwickelt, welches wiederum auf dem Java Compiler Framework Projekt names Polyglot [15] und dessen Softwarearchitektur basiert. Für X10i wird Firm für Zwischendarstellung, Optimierung und Codegenerierung eingesetzt. Während der X10-Compiler von IBM Quellcode in C++ oder Java generiert, erzeugt X10i mit dem Firm-Backend direkt Binärcode für verschiedene von Firm unterstützte Plattformen.

2.5. bytecode2firm

Zu Beginn dieser Arbeit wurde nicht direkt X10i als Compilerfrontend zum Testen der Implementierung verwendet, sondern zunächst ein Projekt namens bytecode2firm. Dabei handelt es sich um einen Compiler der Java-Bytecode mithilfe von Firm in Binärcode übersetzt.

Das Vorgehen von bytecode2firm ist stark an die des GCJ-Projekts der GNU Compiler Collection [16] angelehnt.

Das Bytecode2firm-Projekt ist in einem relativ frühen Entwicklungsstadium, da es bisher nur nebenbei als Proof-of-Concept am Lehrstuhl entwickelt wurde. Es ist dennoch in der Lage viele Java-Programme in lauffähigen Binärcode zu übersetzen. Einige Sprachfeatures wie beispielsweise Exceptions sind derzeit nicht vollständig implementiert. Auch kann bisher nur ein Hauptprogramm erzeugt werden, keine Bibliotheken. Als Laufzeitumgebung und Standardbibliothek wird entweder die libgcj oder eine angefangene eigenentwickelte Variante davon verwendet.

2.6. Verwandte Arbeiten

Die vorliegende Arbeit baut auf die Arbeiten von Bacon [1][2] auf und beschreibt die Implementierung der RTA für das Compilerframework Firm.

Es gibt einige Arbeiten zur Optimierung von dynamisch gebundenen Aufrufen in modernen Laufzeitumgebungen mit JIT-Compiler. Diese Verfahren zur Laufzeit unterscheiden sich allerdings deutlich von statischen Verfahren wie der RTA, dadurch dass sie die Möglichkeiten von dynamischen Optimierungen nutzen können.

Es gibt ein Paper von Namolaru [4] welches über die Implementierung von Devirtualisierung mittels RTA für die GNU Compiler Collection berichtet. Dort wird zudem eine interessante Erweiterung beschrieben, die die RTA um eine einfache Datenflussanalyse erweitert, um die Menge der lebendigen Typen in unterschiedlichen Teilen des Codes weiter einzuschränken. Die Datenflussanalyse berechnet dabei, wie die Typen innerhalb des Codes weitergegeben werden und lässt damit Schlussfolgerungen zu, dass manche Typen bestimmte Aufrufe nicht erreichen können und damit deren Methoden als Zie-

le nicht in Frage kommen können. Diese Erweiterung konnte aus Zeitgründen in der vorliegenden Arbeit nicht vorgenommen werden.

Für mehr Genauigkeit gibt es weitere statische Analysen. Im C++-Umfeld werden auch die Ergebnisse von Alias-Analysen für die Devirtualisierung herangezogen. Eine Alias-Analyse bestimmt, ob sich Speicherbereiche von verschiedenen Referenzen überschneiden können oder nicht. Alias-Analysen sind allerdings sehr aufwendig. Somit lohnt sich dieses Vorgehen nur wenn man aus anderen Gründen die Alias-Analyse macht und diese mit der Optimierung der dynamisch gebundenen Aufrufe koppeln möchte.

Zudem gibt es die Points-To-Analysis [17] in verschiedenen Varianten. Sie bestimmt direkt welche Referenzen auf welche Objekte zeigen können, indem die Zuweisungen verfolgt werden. Die Points-To-Analysis ist wesentlich genauer als die RTA. Sie kann fast alle Fälle entdecken, in denen die dynamische Bindung nicht benötigt wird. Allerdings ist sie auch deutlich aufwendiger (im Allgemeinen $O(n^3)$). Es gibt einige Forschung dazu wie Points-To-Analysis mit Kompromissen bei der Genauigkeit [18] oder durch selektiven Einsatz effizienter gemacht werden kann.

3. Entwurf und Implementierung

In diesem Kapitel werden wichtige Aspekte von Entwurf und Implementierung beschrieben. In Abschnitt 3.1 werden die grundlegenden Entwurfsentscheidungen besprochen. In Abschnitt 3.2 wird die Repräsentation von dynamisch gebundenen Methodenaufrufen in Firm erklärt und wie die Analyse und Devirtualisierung daran erfolgen kann. In Abschnitt 3.3 wird erklärt, wie die RTA in Firm die Objekterzeugung erkennen kann und in Abschnitt 3.4 wird vorgestellt, wie die Zielmethoden von dynamisch gebundenen Aufrufen bestimmt werden. Danach wird in Abschnitt 3.5 besprochen, wie die Voraussetzungen der RTA eingehalten werden können und wo man dabei auf Grenzen stößt.

3.1. Entwurfsentscheidungen

Als Erstes musste geklärt werden, in welcher Softwareschicht der Firm-Verarbeitung, der RTA Code platziert werden soll: In der grundlegenden Bibliothek libFirm, in der Zusatzbibliothek für Objektorientierung liboo oder, wenn es nicht anders ginge, direkt im Frontendcode des Compilers X10i auf hoher Ebene? Wünschenswert ist so tief wie möglich, da der Code dann flexibler für mehr Projekte genutzt werden kann.

Da die RTA allerdings auf objektorientierten Konzepten arbeitet, kann sie zwangsläufig nicht auf der niedrigen Abstraktionsebene von libFirm funktionieren. Klassen und deren Vererbungsbeziehungen können zwar rein mit den Mitteln von libFirm repräsentiert werden, die Repräsentation von dynamisch gebundenen Aufrufen wurde aber in die liboo verlagert. Desweiteren fehlen einige objektorientierte Zusatzinformationen, die nur mit liboo verwaltet werden. Darüberhinaus können Compiler von nicht-objektorientierten Sprachen, welche ohne liboo entwickelt werden, die RTA nicht sinnvoll einsetzen, da diese Sprachen keine Möglichkeit besitzen Subtyp-Beziehungen und dynamische Bindung ausreichend abstrakt auszudrücken. Als objektorientierte Programmanalyse und Optimierung von dynamischer Bindung passt die RTA aber gut in die Konzeption von liboo.

Als Teil der Bibliothek liboo muss die Implementierung der RTA möglichst flexibel mit jedem denkbaren Compilerprojekt, das die Bibliothek benutzt, funktionieren und daher unbedingt sprach- und frontendunspezifisch sein. Das bedeutet, dass keinerlei Details der Sprachen und der Compilerprojekte direkt unterstützt werden sollen. Stattdessen soll die RTA auf frontendunspezifischen Firm-Knoten und Konzepten arbeiten. Ist es erforderlich, die Verarbeitung an frontendspezifische Details anzupassen, so müssen gegebenenfalls Mechanismen, wie z.B. Callbacks, eingeführt werden.

Vorgabe war, die RTA möglichst durch einen einzelnen Aufruf zur Durchführung der Analyse und Optimierung vom Frontend aus einsetzbar zu machen. Insbesondere sol-

len keine komplexen Datenstrukturen gefüllt mit den Analyseergebnissen ins Hauptprogramm zurückgegeben werden. Dies würde die Komplexität erhöhen und eine Übereinkunft über die verwendeten Container-Datenstrukturen erfordern, was so nicht möglich war. Der Nachteil ist allerdings, dass die Analyseergebnisse nicht für weitere Zwecke außerhalb des RTA-Codes verwendet werden können. Die Analyse und die Optimierungen der RTA können so nur gebündelt und isoliert genutzt werden.

Die RTA durchläuft allen Code des zu analysierenden Programms, der vom Hauptprogramm bzw. generell von den Eintrittspunkten aus aufgerufen werden kann. Das entspricht dem Aufbau eines Callgraphen. In der Arbeit von Bacon wird beschrieben, dass ein zuvor erstellter Callgraph für die RTA genutzt wird. Dieser Callgraph ist in Graphenform als Datenstruktur aber nicht notwendig für die RTA. Er kann wie beschrieben auch nicht weiterverwendet werden. Somit reicht es aus, den Code in vergleichbarer Weise entlang der Methodenaufrufe zu durchlaufen, aber den Callgraphen nicht konkret zu speichern.

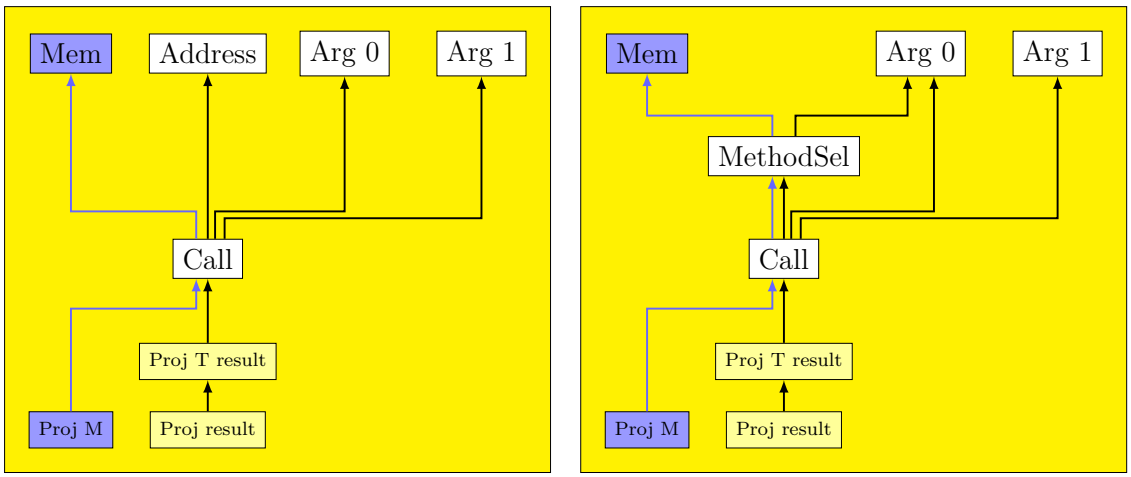
Aufgrund des fluss- und kontextinsensitiven Verhaltens der RTA spielt es keine Rolle in welcher Reihenfolge die aufgerufenen Codeteile analysiert werden. Wichtig ist nur, dass aller lebendiger Code einmal von der Analyse durchlaufen wird. Dies legt einen Algorithmus mit einer Workqueue nahe. Angefangen von den Eintrittspunkten, werden nach und nach alle zu analysierenden Methoden, in der Reihenfolge wie die Aufrufe bei der Analyse gefunden werden, in die Workqueue hinzugefügt und schließlich nacheinander abgearbeitet. Bereits analysierte Methoden werden als erledigt markiert um wiederholtes Analysieren zu verhindern. Um eine Methode zu analysieren, wird ihr Firm-Programmgraph durchlaufen und alle gefundenen Methodenaufrufe und Objekterzeugungen behandelt.

Bei dem beschriebenen Vorgehen wird erst nach und nach bekannt, welche Klassen lebendig sind und welche nicht. Dadurch werden bei dynamisch gebundenen Aufrufen zunächst einige Zielmethoden nicht in den Durchlauf miteinbezogen und analysiert, auch wenn sich später herausstellt, dass sie doch benötigt werden, da die zugehörige Klasse lebendig geworden ist. Deshalb wird ein Update-Vorgang notwendig (wie bereits in [2] zu finden), der jedesmal, wenn eine neue Klasse als lebendig erkannt wird, die zugehörigen Methoden nachträglich in die Ergebnismengen der Aufrufe aufnimmt, als lebendig markiert und sie für die Analyse in die Workqueue einreiht. Eine detailliertere Beschreibung ist in Abschnitt 3.4 zu finden.

Da die RTA die Ergebnismengen optimistisch aufbaut, ist das Ergebnis erst nach einem vollen Durchlauf gültig. Deshalb ist für die Devirtualisierungen ein zweiter Durchlauf notwendig, bei dem der Programmcode noch einmal in gleicher Weise durchlaufen wird.

3.2. Devirtualisierung in FIRM

Hauptzweck der RTA ist die Devirtualisierung dynamisch gebundener Aufrufe. In diesem Abschnitt wird deren Repräsentation in Firm erklärt und wie die Analyse und Devirtualisierung daran erfolgen kann.



(a) statisch gebundener Aufruf (der Normalfall in FIRM)

(b) dynamisch gebundener Aufruf mit Objektreferenz als erstem Argument (Arg 0)

Abbildung 3.1.: Varianten der Methodenaufrufe

Funktionsaufrufe[1] werden in FIRM durch einen `Call`-Knoten repräsentiert. Dieser hat als Operanden die, wegen möglicher Seiteneffekte nötige, Speicherkante `Mem`, einen Knoten der die aufzurufende Funktion angibt und gegebenenfalls n weitere Knoten für n Argumente. Statisch gebundene Aufrufe sind mit einem `Address`[2]-Knoten als Aufrufziel verbunden, der einen Verweis auf die Entity einer Funktion in sich trägt und damit direkt die aufzurufende Funktion angibt. Dynamisch gebundene Funktionsaufrufe sind an dieser Stelle statt mit einem `Address`-Knoten mit einem `MethodSel`[3]-Knoten verbunden. Dieser enthält ebenfalls eine Entity einer Funktion – hier im Weiteren *Call Entity* genannt – welche nun aber nur als Ausgangspunkt für die Suche nach der aufzurufenden Methode dient. Schematische Abbildungen beider Aufrufvarianten sind in Abbildung 3.1 dargestellt.

Für die Devirtualisierung will man einen dynamisch gebundenen Aufruf in einen statisch gebundenen verwandeln. Dazu ersetzt man den `MethodSel`-Knoten am `Call`-Knoten durch einen `Address`-Knoten mit der Entity der einzigen Zielmethode.

Die Call Entity charakterisiert einen dynamisch gebundenen Aufruf. Sie gibt eine Klasse und den Namen der aufzurufenden Methode an. Die Klasse dient für den Aufruf als eine Art Obergrenze in der Klassenhierachie, alle Unterklassen davon können Typ des angesprochenen Objekts sein und gegebenenfalls eigene Implementierungen der Methode besitzen, die von dem Aufruf zur Laufzeit angesprungen werden können.

Da für die RTA im Grunde nur wichtig ist, welche Methoden aufgerufen werden und von welcher Klasse aus die Suche nach potentiellen Zielmethoden beginnt, aber nicht

[1] Da FIRM auf C-Ebene bzw. tiefer angesiedelt ist, wird hier die C-Nomenklatur verwendet. Dabei ist mit Funktion ein Low-Level-Konstrukt gemeint, welches sowohl objektorientierte Methoden alsauch C-Funktionen implementieren kann.

[2] in früheren Versionen `SymConst`

[3] in früheren Versionen `Sel`

welche Methode welche aufruft, kann man alle Methodenaufrufe im Programm mit derselben Call Entity zusammenfassen. Sie haben alle die gleiche Menge an Zielmethoden. Das geht darauf zurück, dass letztendlich nur zwei Faktoren Einfluss auf die Berechnung der Menge haben: die Klassenhierarchie und die Menge der lebendigen Klassen im Programm. Beides ist für gleichlautende Call Entities verschiedener Aufrufe im Programm gleich. Die Unterscheidung nach Aufrufstellen und aufrufenden Funktionen, die Bacon in seiner Arbeit [2] macht, ist deshalb nicht notwendig, solange man diese Information nicht noch für andere Zwecke braucht.

3.3. Erkennung der Objekterstellung

Die RTA muss erkennen können welche Objekte im Programm erstellt werden. Die RTA nach Bacon erkennt die Objekterzeugung an Konstruktoraufrufen. Dies bringt einige Probleme mit sich, u.a. dass Konstruktoraufrufe von Oberklassen zur Konstruktion von Subobjekten im Konstruktor einer Klasse nicht gezählt werden dürfen.

Das Konzept von Konstruktoren ist allerdings in liboo und libFirm nicht vorhanden. Das hat vorallem zwei Gründe: Für Konstruktoren gibt es keine eindeutige Definition, ihre Semantik ist sprachabhängig, und das Konzept Konstruktor ist in der Zwischendarstellung und im Backend nicht unbedingt nötig, denn im Grunde kann eine beliebige Funktion den Zweck der Initialisierung eines Objekts erfüllen. In Firm werden Konstruktoren also auf normale Funktionen abgebildet.

Die Objekterzeugung kann in Firm unterschiedlich aussehen, je nach Semantik der Quellsprache bzw. Implementierung des Frontends. Für die vorliegende Arbeit wird davon ausgegangen, dass sie ungefähr folgendem Muster entspricht. Zunächst wird der Speicherplatz für das Objekt allokiert, typischerweise über einen Aufruf einer Allokationsfunktion, im einfachsten Fall die externe C-Funktion `calloc`. Danach wird ein spezieller Zeiger im Objekt – der *VPointer* – auf eine spezielle Tabelle der Klasse – *VTable* genannt – initialisiert, welche Informationen zur Abwicklung der dynamischen Bindung zur Laufzeit enthält. Im Anschluss wird zur restlichen Initialisierung eine Funktion für das Objekt aufgerufen, welche den Code des Konstruktors enthält. Danach ist das Objekt voll initialisiert und kann verwendet werden.

Die Frage ist, ab wann ist ein Objekt ein Objekt? Tatsächlich lässt sich argumentieren, dass der schärfste Übergang die VPointer-Initialisierung ist. Davor ist es nur ein allokierter Speicherbereich, bestenfalls ein Objekt unbestimmten Typs. Danach ist es zwar noch nicht vollständig initialisiert (u.a. Invarianten noch nicht gültig), aber es ist nachweislich ein Objekt des Typs entsprechend dem VTable (bzw. weiterer damit verbundener Datenstrukturen), auf den der VPointer des Objekts verweist. In den meisten Implementierungen wird die Typbestimmung zur Laufzeit über den VPointer abgewickelt. Somit könnte man sagen, dass dieser dem Objekt den Typ verleiht. Desweiteren ermöglicht ein initialisierter VPointer grundsätzlich auch den Aufruf dynamisch gebundener Methoden auf dem Objekt. Das bedeutet, ab dem Zeitpunkt, an dem der VPointer gesetzt ist, kann das Objekt als solches mit seinen Methoden benutzt werden (auch wenn das in seman-

tisch gültiger Weise erst nach der Initialisierung durch den Konstruktorcode stattfinden kann).

Damit bietet sich für die Erkennung der Objekterstellung in Firm die VPointer-Initialisierung an. Nebenbei löst das Erkennen der Objekterzeugung anhand der VPointer-Initialisierung auch das Problem mit den Konstruktoraufrufen von Oberklassen bei der Konstruktion von Subobjekten, welches bei der Erkennung anhand des Konstruktoraufrufs besteht.

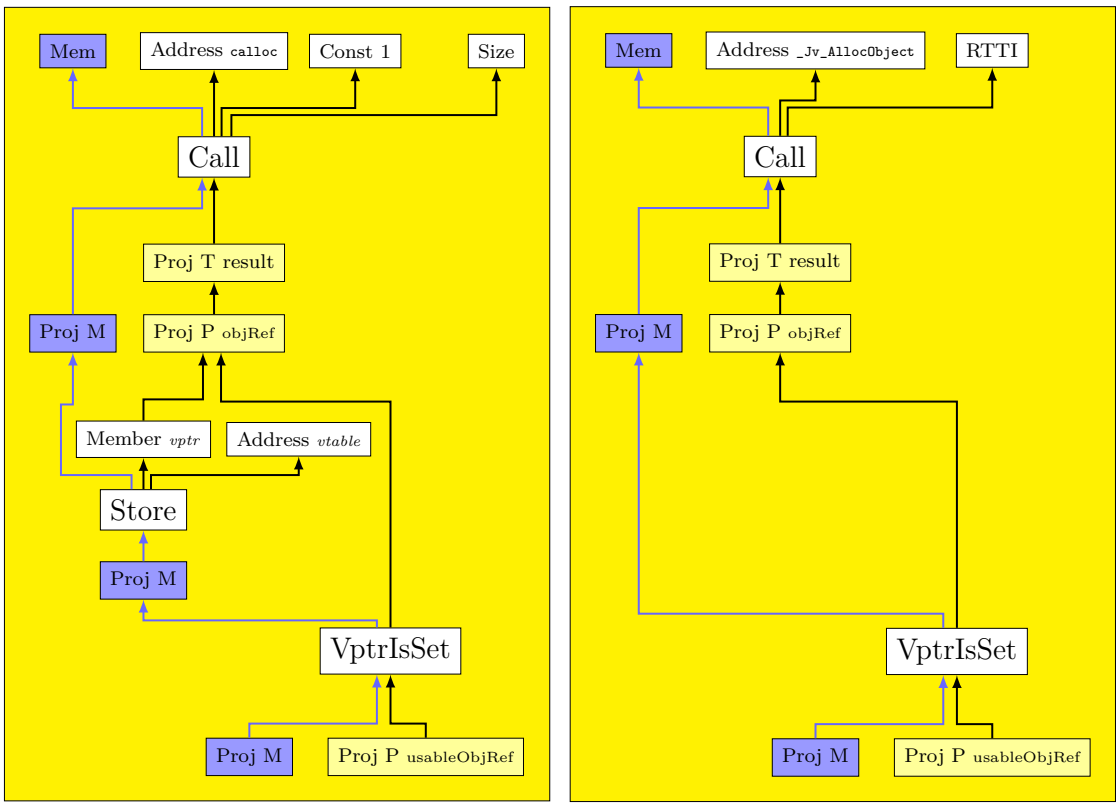
Die naheliegende Variante in Firm den VPointer, welcher ein normales Feld[4] jeder Klasse ist, zu initialisieren, ist durch einen `Store`-Knoten. Das Problem ist aber, dass es andere Varianten geben kann. Bei bytecode2firm trat der Fall auf, dass in neueren Versionen eine frontendspezifische Allokationsfunktion (`_Jv_AllocObject`) verwendet wird, welche nicht nur den Speicherplatz für das Objekt allokiert sondern auch gleich noch den VPointer initialisiert. Dieses Vorgehen wurde vom Vorbild GCJ übernommen. In diesem Fall ist das Setzen des VPointers nicht im Firm-Graphen erkennbar sondern bleibt in der externen Funktion verborgen.

Die erste Idee war für grundlegende, objektorientierte Konzepte wie Objekterzeugung (und z.B. auch Klasseninitialisierung) abstrakte Firm-Knoten einzuführen, die zunächst in einer abstrakten OO-Phase als informationstragende Platzhalter dienen und die dann beim OO-Lowering durch die jeweilige frontendspezifische Implementierung ersetzt werden, genauso wie es bereits bei dynamisch gebundenen Aufrufen mit dem Knoten `MethodSel` gemacht wird. Dies erwies sich jedoch als schwierig umzusetzen, da für Firmknoten eine sehr präzise Definition gefordert wird, welche in diesem Fall nicht möglich war, da die genauen Semantiken der objektorientierten Konzepte unter Umständen sprachabhängig sind.

Infolgedessen wurde ein neuer Firm-Knoten namens `VptrIsSet` eingeführt. Er wird zwischen die wie auch immer geartete VPointer-Initialisierung und die weitere Verwendung der Objektreferenz gehängt und fungiert als Markerknoten. Das bedeutet er steht nicht für eine Operation sondern platziert lediglich zusätzliche Information an eine bestimmte Stelle im Programmgraphen. Beim OO-Lowering wird dieser Knoten dann wieder entfernt. Der `VptrIsSet`-Knoten hat als Eingänge neben der zu markierenden Objektreferenz auch die Speicherkante, was notwendig ist, um ihn in der Reihenfolge exakt hinter der VPointer-Initialisierung zu platzieren. Auf diese Weise markiert der `VptrIsSet`-Knoten präzise, ab wann der VPointer gesetzt ist und er enthält einen Verweis auf den Typ des neuen Objekts. Durch dieses Vorgehen funktioniert es auch problemlos mit frontendspezifischen Methodenaufrufen oder anderen denkbaren Varianten. Was auch immer getan wird um das neue Objekt zu allokieren und den VPointer zu initialisieren, das Frontend kann den `VptrIsSet`-Knoten dahinter in den Graph einbauen um auf frontendunabhängige Weise für objektorientierte Analysen klarzustellen was passiert. Damit stellt der neue Knoten eine gute Lösung für die RTA dar, um die Objekterzeugung zu erkennen.

In Abbildung 3.2 ist die Verwendung des `VptrIsSet`-Knotens beispielhaft gezeigt, einmal in der Variante mit explizitem Setzen des VPointers durch einen `Store`-Knoten wie

[4] entsprechend Java-Terminologie



(a) Objekterzeugung mit `Store` VPointer und `VptrIsSet`

(b) Objekterzeugung mit frontendspezifischer Funktion und `VptrIsSet`

Abbildung 3.2.: Verwendung des Markerknotens `VptrIsSet`

es in X10i vorkommt und einmal bei Verwendung einer frontendspezifischen Allokationsfunktion, die nach der Allokation auch den VPointer initialisiert, wie in bytecode2firm. Der Wert, der durch den `Proj P`-Knoten unterhalb des `VptrIsSet`-Knotens repräsentiert wird, ist dann die Objektreferenz, welche im Weiteren genutzt werden sollte.

3.4. Finden der möglichen Zielmethoden

Um zu entscheiden, ob ein Aufruf devirtualisiert werden kann, bestimmt die RTA die Menge der möglichen Zielmethoden der dynamisch gebundenen Aufrufe, die im Programm vorkommen.

Trifft die RTA bei ihrem Durchlauf auf einen dynamisch gebundenen Methodenaufruf, so muss entschieden werden, welche der Zielmethoden tatsächlich aufgerufen werden können. Nur diese werden zu dem von der RTA reduzierten Callgraph hinzugefügt. Sie werden als lebendig markiert und zur Analyse in die Workqueue eingefügt. Um die Zielmethoden einzusammeln, wird die Klassenhierarchie anhand der im Aufruf angegebenen Call Entity durchsucht. Sie gibt den Namen der Methode und die Klasse an, die als Ausgangspunkt der Suche in der Klassenhierachie dient.

Laut Theorie ist zuerst ein statischer Lookup nach oben in der Klassenhierarchie nötig, um im Falle einer ererbten Methode die Implementierung in der jeweiligen Oberklasse zu finden, und danach eine Suche abwärts in allen Unterklassen, um überschreibende Methoden einzusammeln.

In Firm können für ererbte Methoden Entity-Kopien in Unterklassen platziert werden. Dies ist aus zwei Gründen sinnvoll. Erstens, um Methodenaufrufe so repräsentieren zu können wie sie auch im Quellcode vorkommen, mit dem korrekten statischen Typ der Referenz des Aufrufs, ist die Existenz einer entsprechenden Entity erforderlich, die dann von einem `Address`-Knoten referenziert werden kann. Wird stattdessen die zugehörige, geerbte Methoden-Entity der Oberklasse referenziert, wird der Aufruf zu einem abstrakteren Aufruf verändert, der fälschlicherweise mit der Oberklasse als statischem Typ in Verbindung gebracht wird. Das ist mit einem Informationsverlust verbunden, der die Optimierung durch die RTA in bestimmten Fällen verhindern kann. Zweitens ist eine Entity-Kopie sinnvoll in den speziellen Fällen, dass z.B. in Java eine Klasse ein Interface implementiert, aber für eine (oder mehrere) der Interface-Methoden nicht selbst eine Implementierung bereitstellt sondern diese von einer Oberklasse erbt. Um diese Dreiecks-zuordnung festzuhalten, ist es sinnvoll, die Entity-Kopie in der Unterklasse zu platzieren. Legt das Frontend diese Zuordnung nicht fest, ist diese unter Umständen vom frontendunabhängigen Bibliothekscode nicht ohne sprachspezifische Annahmen wiederzufinden. Zudem ist die Suche danach mit erhöhtem Aufwand für die Analysen verbunden.

Zu einer Entity-Kopie kann das zugehörige Original abgefragt werden. Bei der Abwärtssuche müssen die Entity-Kopien allerdings speziell behandelt werden um nicht fälschlicherweise die nicht voll funktionsfähigen Kopien einzusammeln, sondern diese nur als Verweise auf die eigentlichen Methodenimplementierungen zu nutzen.

Der anfängliche statische Lookup ist in Firm entweder schon gemacht (bei fehlenden Entity-Kopien) oder passiert mithilfe der Entity-Kopie automatisch am Anfang der Abwärtssuche.

Die Abwärtssuche läuft rekursiv von Klasse zu Unterklasse alle Zweige der Klassenhierarchie ab. Sie findet für jede Klasse die zugehörige Implementierung der Methode anhand des Namens der Entity, welcher die Signatur enthalten sollte. Falls die Klasse lebendig ist, wird die Methode in die Zielmenge der Call Entity eingefügt. Abstrakte Methoden werden dabei ignoriert.

Zielmethoden von nicht-lebendigen Klassen werden nicht einfach verworfen, sondern müssen bei dem verwendeten Workqueue-Algorithmus ebenfalls separat gesammelt und bereit gehalten werden, da die zugehörigen Klassen auch später noch lebendig werden können, wenn während des weiteren Durchlaufs eine entsprechende Objekterzeugung im Programmcode gefunden wird.

Der Workqueue-Algorithmus durchläuft die einzelnen Methoden theoretisch in beliebiger Reihenfolge, was für die RTA keinen Unterschied macht. Die tatsächliche Reihenfolge ergibt sich daraus, dass die Methoden in der Reihenfolge, wie sie im Code gefunden werden, in die Workqueue eingereiht und schließlich abgearbeitet werden.

Die aussortierten Methoden von nicht-lebendigen Klassen werden in einer verschachtelten assoziativen Datenstruktur gespeichert, so dass zu einer Klasse die Zuordnungen von Methoden zu je einer Menge von Call Entities effizient wiederzufinden sind.

Wird eine Klasse lebendig, wird ein Update-Vorgang durchgeführt. Es werden anhand der Datenstruktur alle zurückgehaltenen Methoden der Klasse zu den Zielmengen der jeweiligen Call Entities hinzugefügt, um den Zustand der Analyseergebnisse auf einen korrekten Stand zu bringen.

Bei Firm gibt es noch eine weitere Besonderheit bei Methoden-Entities zu beachten. Entities besitzen zwei Namen, einen normalen Namen für die interne Nutzung in der Compilerlogik und einen Linkernamen. Letztere ist der Name, den das jeweilige Programmkonstrukt am Ende im Binary erhält beziehungsweise auf welchen Verweise auf diese Entity im Binary verweisen. Damit ist es auch möglich Entities als Weiterleitungen auf andere Funktionen zu nutzen, wenn sie den selben Linkernamen wie das Ziel zugewiesen bekommen. Der normale Name ist dann nur noch eine Art Alias.

Die RTA-Implementierung muss dieser Art der Weiterleitung folgen können, um allen Code analysieren zu können. Besitzt eine Methode keinen Graphen, dafür aber einen geänderten Linkernamen, zu dem eine andere Funktion existiert, so wird diese ebenfalls lebendig markiert und analysiert.

3.5. Einhaltung der Voraussetzungen

Die RTA setzt wie in Abschnitt 2.2 beschrieben voraus, dass sie allen Code, den das Programm ausführen könnte, analysieren kann. Ausnahmen sind nur für erkennbar abgegrenzte und separat entwickelte Bibliotheken möglich. Zusätzlich benötigt die RTA das Wissen über die vollständige Klassenhierachie. Diese Anforderungen sind in der Praxis aber nicht immer zu erfüllen.

Native Funktionen, die insbesondere auch im Laufzeitssystem für hardware- oder systemnahe Funktionalität gebraucht werden, sind in einer anderen Programmiersprache (meist C) implementiert und sind im Normalfall nicht zur Übersetzungszeit eines Programms als Firm-Graph verfügbar. Stattdessen wird deren Code am Ende nur dazugelinkt. Trotzdem entspricht das oft nicht dem Fall einer Bibliothek, den die RTA handhaben kann. Die nativen Funktionen sind für die RTA erstens nicht immer als zu einer Bibliothek zugehörig erkennbar, es sind in Firm einfach nur Funktionen ohne Graph, und zweitens können sie u.U. programmintern beliebige Methoden aufrufen und Objekte erstellen. Eine separat entwickelte Bibliothek kennt dagegen die programminternen Methoden und Klassen nicht und greift auf diese nicht beliebig zu. Da die RTA die nativen Funktionen nicht analysieren kann, fehlt ihr die Information darüber was diese Funktionen tun, also welche Funktionen aufgerufen werden und was für Objekte erstellt werden. Unter diesen Bedingungen kann die RTA normalerweise nicht korrekt arbeiten.

Allerdings ist es unter Umständen möglich, die fehlenden Informationen auf anderem Weg der Analyse bekannt zu machen. Die Korrektheit der von der RTA durchgeführten Devirtualisierungen hängt nur davon ab, ob die RTA weiß, welche Klassen tatsächlich lebendig sind und ob sie die Klassenhierarchien vollständig kennt und dass diese sich zur Laufzeit nicht ändern. Schafft man es trotz nicht-analysierbaren Codeteilen (die nicht als separat entwickelte Bibliothek abgegrenzt sind), den für das jeweilige Programm korrekten Informationstand zu erreichen, kann die RTA damit prinzipiell korrekt arbeiten.

Eine Möglichkeit besteht darin, Klassen und Methoden, die in nicht-analysierbarem Code instanziiert und aufgerufen werden, zu den Initialmengen hinzuzufügen und so der Analyse bekannt zu machen. Da die RTA die Information über lebendige Funktionen und Klassen global betrachtet, macht dieses Vorgehen im Grunde keinen Unterschied zum Finden während der Analyse. Dies funktioniert prinzipiell gut für nativen Code des Laufzeitsystems und der Standardbibliothek, da sie dem Compilerentwickler bekannt sind und die Menge der in diesen nativen Codeteilen instanziierten Klassen gering ist und sich üblicherweise selten ändert.

Dieser Ansatz funktioniert allerdings nicht bei Code, der in Abhängigkeit des zu übersetzenden Programms generiert wird, etwa eine native Methode, die in Abhängigkeit von Argumenten unterschiedliche Methoden aufruft. Ein Beispiel dafür ist die Klasseninitialisierung in bytecode2firm nach dem Vorbild des GCJ-Projekts. Der Aufruf der nicht-analysierbaren, nativen Funktion `_Jv_InitClass` verdeckt den Aufruf der Initialisierungsmethoden der unterschiedlichen Klassen, welche die GCJ-Funktion anhand der Informationen in der RTTI-Datenstruktur findet, welche ihr als Argument übergeben wird. Die Analyse kann das frontendspezifische Konstrukt zur Klasseninitialisierung nicht behandeln. Zudem steht die RTTI-Datenstruktur zum Analysezeitpunkt nicht zur Verfügung. Die Menge der über diese Konstrukte aufgerufenen Klasseninitialisierungsfunktionen ist veränderlich und abhängig vom zu übersetzenden Programm. Das Frontend könnte sich höchstens merken, für welche Klassen es die Klasseninitialisierungsfunktionen aufbaut und diese dann der RTA in Initialmengen mitteilen. Dann hätte man aber die Reduzierung auf die tatsächlich genutzten Codeteile nicht.

Weitere Möglichkeiten sind das Bereitstellen der Informationen in einer Datenstruktur durch das Frontend oder eine Abfrage der Information vom Frontend, z.B. durch Callbacks. Letzteres ist derzeit als frontendunabhängiger Workaround erforderlich für den beschriebenen Fall der GCJ-Klasseninitialisierung bei bytecode2firm.

Definiert der Programmierer native Methoden im Programm selbst, kann auch der Compilerentwickler die Information nicht liefern. Im Grunde wäre dann die Korrektheit der RTA auf die Mithilfe des Benutzers angewiesen. Das Frontend könnte eine Eingabemöglichkeit, etwa als zusätzliches Argument, auf der Kommandozeile anbieten. Da der Nutzer dazu aber genauer über die RTA Bescheid wissen müsste und die Informationen über Aufrufe und Instanziierungen in seinem nativen Code selbst zusammentragen müsste, ist das wenig praktikabel.

Grenzen setzen auch das heutzutage verbreitete Nachladen von Klassen zur Laufzeit und die generische Objektinstanziierung via Reflection, z.B. in Java mit den Methoden `Class.forName` und `Class.newInstance`. Durch dynamisches Nachladen können in der Klassenhierarchie unterhalb beliebiger nicht-finaler Klassen neue, unbekannte Unterklassen auftauchen. Sich zur Laufzeit ändernde Klassenhierachien würden aber wichtige Grundannahmen der CHA und RTA zunichte machen, da diese Analysen vorallem aus der Klassenhierarchie ihre Schlüsse ziehen. Dynamisches Nachladen und generische Objektinstanziierung kann von der RTA als rein statische Analyse nicht unterstützt werden.

Für die vorliegende Arbeit wurden diese im Zusammenhang mit der RTA kaum lösbaren Fälle vernachlässigt. In X10 gibt es derzeit weder Reflection noch dynamisches Nachladen von Klassen.

4. Evaluation

In diesem Kapitel wird untersucht, wie sich die Optimierung mit der erstellten RTA-Implementierung auswirkt. Zu diesem Zweck wurden Laufzeitmessungen gemacht. Zur Anschauung wurden auch die Anzahl der Aufrufe und der Devirtualisierungen der Testprogramme gezählt. Die Zählung liefert allerdings keine neuen Erkenntnisse, da die RTA-Implementierung für Firm direkt auf der klassischen RTA von Bacon [1][2] basiert und somit keine besseren Ergebnisse erzielen kann.

In erster Linie geht es um den Performancegewinn durch die Devirtualisierung bei den Laufzeiten von kompilierten Programmen. Dieser wird in Abschnitt 4.1 untersucht. Daneben ist aber auch interessant wie sich die Compilerlaufzeit verändert. Da die RTA das gesamte Programm analysiert, ist darauf zu achten, dass der Analyselauf nicht zu teuer wird. Dies wird in Abschnitt 4.2 überprüft.

Die Testumgebung für die Laufzeitmessungen war ein Intel Core i3 550 mit 3.20 GHz, 4 GB RAM (ca. 3,7 GB nutzbar) unter 64 Bit Ubuntu 14.04.2 LTS mit Linux Kernel 3.13.0-24-generic. Gemessen wurde mit den Softwareversionen: X10i 2dcc035[1], liboo bbd10d6[2] und libFirm 80f9638[3].

Für alle Tests wurde mit X10i stets für die Plattform *i686-invasic-irtss* kompiliert. iRTSS steht für *Invasive Run-Time Support System* und ist die invasive Laufzeitumgebung und System API, die am Lehrstuhl hauptsächlich verwendet wird. In diesem Fall handelt es sich um die Variante, die auf x86 Linux läuft.

4.1. Optimierungserfolg

Um den Optimierungserfolg zu untersuchen, wurden mehrere Programme einmal ohne RTA, einmal mit RTA, einmal mit RTA plus zusätzlicher Inlining-Optimierung und einmal nur mit Inlining kompiliert, ausgeführt und ihre Laufzeit gemessen.

Leider stand für die Evaluation kein großes objektorientiertes Programm mit vielen dynamisch gebundenen Aufrufen zur Verfügung, wie man es sich intuitiv vorstellen würde oder wie beispielsweise Bacon und Sweeney in [1] evaluieren. Andererseits ist es nicht zwingt erforderlich, da die Wirksamkeit der klassischen RTA nicht nochmal nachgewiesen werden muss.

Der wichtigste Teil der vorliegenden Arbeit befasste sich mit X10. X10 ist eine Programmiersprache, die bisher noch wenig verbreitet ist. Darum gibt es nur relativ wenig verfügbaren Code. Darüberhinaus wird X10 vornehmlich für numerische Anwendungen

[1] `2dcc0356a7578b759b46ef773fd324963524cce7`
[2] `bbd10d6ec56212258ff9e3ffb2f3dd33f663c5e6`
[3] `80f9638213f614b9321047b9f66effb03dca95ca`

genutzt, welche typischerweise Objektorientierung weniger oder gegebenenfalls in anderer Weise nutzen als typische moderne objektorientierte Software. Allerdings ist X10 selbst, durch seine Intention, die Produktivität zu erhöhen, und seine Verwandtschaft zu Java, sehr objektorientiert ausgerichtet, was Sprachmittel, wie z.B. foreach-Schleifen, die Standardbibliothek oder auch die Umsetzung von Operatoren, Closures usw. angeht.

Dass kein wirklich großes Programm getestet werden konnte ist insofern tragisch, als dass kleine Programme mehr bzw. leichter von der RTA profitieren können als große. Die ist prinzipbedingt, weil die RTA eine Whole-Program-Analysis ist und die Information, welche Klassen genutzt werden, global für das gesamte Programm betrachtet. Kleine Programme nutzen üblicherweise weniger Klassen wodurch die Wahrscheinlichkeit höher ist, dass zu einem gegebenen dynamisch gebundenen Aufruf nur eine Zielmethode in Frage kommt. Dadurch können mehr Aufrufe devirtualisiert werden.

Um zu sehen, ob die RTA überhaupt etwas tut, ist es naheliegend, möglichst direkt die Auswirkung zu betrachten anhand eines einzigen, optimierbaren, dynamisch gebundenen Aufrufs in einer Schleife, die sehr oft durchlaufen wird (LoopDynCall). Da es mit Interface Calls eine zweite Art von dynamisch gebundenen Aufrufen gibt, die normalerweise mehr Performance kostet, wurde auch eine Variante mit einem Interface Call in einer Schleife betrachtet (LoopInterfaceCall). Hiebei ist zu erwähnen, dass Interface Calls bei X10i sehr teuer sind, da die Implementierung des Lookups in liboo derzeit nicht optimiert ist (z.B. durch Interface Method Tables [19]).

Daneben wurde eine Implementierung des bekannten n-Queens Problems [20] als Testprogramm genommen (NQueensArray). Die Idee war ein rechenintensives Programm zu haben, das viele dynamisch gebundene Aufrufe aufweist. Die Implementierung von n-Queens nutzt üblicherweise Arrays als Datenstruktur und wäre damit in vielen anderen Programmiersprachen nicht objektorientiert. In X10 werden Arrays allerdings über die generische Klasse `Array[T]` implementiert. Dadurch ergeben sich dynamisch gebundene Aufrufe, die zu optimieren sind. Zusätzlich wurde eine zweite Variante (NQueensArrayList) absichtlich verändert, damit sie anstatt der normalen Arrays die Container-Datenstruktur `ArrayList[T]` aus der Standardbibliothek verwendet, und zwar absichtlich über das Interface `List[T]`, damit viele Interface Calls im Code enthalten sind.

Zusätzlich wird ein Programm namens Multigrid herangezogen, welches derzeit als Hauptanwendung des invasiven Compilers X10i zur Erforschung von Invasive Computing genutzt wird [21]. Es implementiert ein Mehrgitterverfahren [22] zur iterativen Berechnung von großen Gleichungssystemen. Anwendungen von Mehrgitterverfahren sind z.B. Berechnungen von Hitzeausbreitung in Metall oder Tsunami-Simulationen. Als Benchmark eingesetzt, berechnet Multigrid standardmäßig eine Simulation der Hitzeausbreitung in einer Metallplatte, auf die ein Laser einen Schriftzug schreibt. Wiederum handelt es sich um ein Programm für numerische Berechnungen, das aber durch die Eigenschaften der Programmiersprache X10 dennoch zu einem gewissen Grad objektorientiert ist. Für die Messung wurde die Version 4c09842[4] von Multigrid verwendet. Für den Parameter `timesteps` von Multigrid wurde ein Wert von 1000 gewählt und die Ausgaben deaktiviert.

[4] `4c098421a853561e5b795e0801af6f0b8e6bbec3`

Die Ergebnisse der Laufzeitmessungen sind in Tabelle 4.1 zu dargestellt. Die Werte der Laufzeiten sind Mittelwerte mehrerer Messungen. Die einzelnen Messwerte sind in Tabellen im Anhang in Abschnitt A.1 angegeben mitsamt einer kurzen statistischen Betrachtung und weiteren Details.

program	default	RTA	RTA+Inlining	only Inlining
LoopDynCall	6,74	6,75	1,10	2,67
LoopInterfaceCall	38,71	6,73	1,10	35,37
NQueensArray	5,19	5,19	1,86	1,87
NQueensArrayList	23,33	7,50	2,02	18,43
Multigrid	32,25	34,31	26,28	30,33

Tabelle 4.1.: Laufzeiten der Kompilate (in Sekunden)

Der Tabelle 4.1 ist zu entnehmen, dass die Devirtualisierung durch die RTA durchaus deutliche Wirkung zeigen kann, zumindest im Zusammenspiel mit Inlining oder bei Interface Calls. Die reine Devirtualisierung von normalen dynamisch gebundenen Aufrufen der Einfachvererbung bringen allerdings keinen messbaren Performancegewinn. Warum sich bei den Testprogrammen LoopDynCall und Multigrid sogar eine leichte Verschlechterung eingestellt hat, bleibt fraglich. An den Ergebnissen der Testprogramme LoopInterfaceCall und NQueensArrayList ist auch deutlich zu sehen, dass die Interface Calls sehr teuer sind, da die liboo derzeit nur über eine unoptimierte Implementierung des Interface Lookups verfügt. Bei Multigrid als einzigem richtigem Anwendungsprogramm fallen die Verbesserungen schwächer aus. RTA mit Inlining schafft aber immerhin ca. 18,5% gegenüber nur ca. 6% bei nur Inlining.

Darüberhinaus ist interessant zu wissen, wieviele Aufrufe tatsächlich devirtualisiert wurden. Gezählt wurden die Anzahl der statisch gebundenen Aufrufe, die Anzahl der dynamisch gebundenen Aufrufe und die Anzahl der devirtualisierten, dynamisch gebundenen Aufrufe. Zusätzlich können weitere Arten von Aufrufen im Code existieren, die hier nicht weiter betrachtet werden. Dabei handelt es sich hauptsächlich um indirekte Aufrufe im C-Code der Laufzeitumgebung, also Aufrufe über Funktionszeiger. Es wurde außerdem zwischen normalen, dynamisch gebundenen Aufrufen der Einfachvererbung und Interface Calls unterschieden. Diese und ihre zugehörigen Devirtualisierungen werden getrennt angegeben, so lassen sich die Zahlen bei Bedarf durch Addieren zusammenfassen.

X10i kompiliert derzeit die gesamte Standardbibliothek bei jedem Compilerlauf mit. Dadurch ergibt sich ein großer Overhead an Aufrufen, den jedes Programm gleichermaßen besitzt. Deshalb wurde zusätzlich ein Programm mit leerer Mainmethode (EmptyMain) ausgewertet um diese Grundmenge an Aufrufen durch den Code des Laufzeitsystems zu bestimmen. Zu beachten ist, dass über diese Grundmenge hinaus verschiedene Programme dennoch verschiedene Anteile an Code aus der Standardbibliothek nutzen. Diese Anteile lassen sich derzeit nicht vom Hauptprogramm trennen und werden hier

zum Programmcode gezählt. Zur besseren Übersicht wurden bei allen Programmen, außer EmptyMain, die Zahlen von EmptyMain abgezogen.

program	static calls	dynamic calls	interface calls	others
EmptyMain	543	94	6	0
LoopDynCall	6	1	0	0
LoopInterfaceCall	6	0	1	0
NQueensArray	199	60	0	0
NQueensArrayList	154	23	12	0
Multigrid	2297	597	87	0

Tabelle 4.2.: Zahl der Aufrufstellen nach Art

program	devirtualizations of dynamic calls		devirtualizations of interface calls	
EmptyMain	93	98,9%	0	
LoopDynCall	1	100%	0	
LoopInterfaceCall	0		1	100%
NQueensArray	60	100%	0	
NQueensArrayList	23	100%	12	100%
Multigrid	545	91,3%	20	23%

Tabelle 4.3.: Zahl der Devirtualisierungen

Die Ergebnisse sind in Tabelle 4.2 und in Tabelle 4.3 zu sehen. An den Zahlen sieht man deutlich, dass die Testprogramme LoopDynCall, LoopInterfaceCall, NQueensArray und NQueensArrayList sehr klein sind und speziell ausgesucht wurden, weil sie die Leistung der RTA besonders herausstellen. Dass 100% der Aufrufe devirtualisiert werden, ist mit anderen und vorallem größeren Programmen in der Praxis nicht zu erwarten. Bei Multigrid kann ein großer Anteil der normalen dynamisch gebundenen Aufrufe devirtualisiert werden, aber nur ein deutlich geringerer Anteil an Interface Calls. Es ist anzunehmen, dass Interface Calls vermehrt an Stellen mit echter Polymorphie eingesetzt werden und Interfaces in größeren Programmen häufig viele lebendige Unterklassen haben.

4.2. Compilerlaufzeit

Um die Änderung der Compilerlaufzeit zu messen, wurden Programme einmal mit und einmal ohne RTA kompiliert und dabei die Zeit gemessen, die der Compiler für die Übersetzung benötigt.

Dazu wurden Multigrid und die erste n-Queens-Variante NQueensArray als Testprogramme gewählt. Multigrid, weil es das größte Programm ist, das zur Verfügung stand, und NQueensArray um einen zweiten Vergleichswert zu haben.

X10i kompiliert derzeit bei der Übersetzung eines Programms stets die gesamte Standardbibliothek mit, was maßgeblich die Compilerlaufzeit verlängert. Da dies von der RTA unabhängig ist und nur die relative Änderung von Bedeutung ist, stellt es kein Problem dar. Der Einfluss auf die Laufzeit der RTA ist geringer, da nur tatsächlich genutzte Teile des Codes analysiert werden. Allerdings ist, wie in Abschnitt 4.1 bereits beschrieben, der Overhead durch den Code des Laufzeitsystems bei jedem Programm gleichermaßen mit dabei.

Die Ergebnisse sind in Tabelle 4.4 dargestellt. Die Werte der Laufzeiten sind Mittelwerte mehrerer Messungen. Die einzelnen Messwerte sind in Tabellen im Anhang in Abschnitt A.1 angegeben mitsamt einer kurzen statistischen Betrachtung und weiteren Details.

program	default	RTA	
NQueensArray	140,88	142,13	$+0,89\%$
Multigrid	206,65	197,87	$-4,25\%$

Tabelle 4.4.: Compilerlaufzeiten von X10i (in Sekunden)

In der Tabelle 4.4 ist zu sehen, dass sich die Compilerlaufzeiten nicht großartig ändern. Für die Übersetzung des größeren Programms Multigrid stellt sich sogar eine Verkürzung der Kompilierzeit ein. Dies könnte bedeuten, dass das Backend aufgrund der Devirtualisierungen Arbeit einsparen kann, und das bereits ohne explizites Wegstreichen von ungenutzem Code anhand der Ergebnisse der RTA. Die Verlängerung der Kompilierzeit bei der Übersetzung von NQueensArray ist statistisch nicht signifikant zu trennen von der Laufzeit ohne RTA, was ein gutes Ergebnis ist. Das bedeutet, dass die RTA die Compilerlaufzeit nicht nennenswert verlängert und unter Umständen, wie bei Multigrid, sogar verkürzt.

Zusätzlich wurde eine isolierte Messung der Compilerphase RTA gemacht. Die Zeitmessung wurde im X10i Code mittels der Java Methode `System.currentTimeMillis` durchgeführt. Die Zeiten vor und nach der RTA wurden genommen, voneinander abgezogen und ausgegeben. Im Gegensatz zu allen anderen Messungen erfolgte damit eine Messung der Realzeiten, im Gegensatz zu reiner CPU-Zeit. Es wurde die Zeit für den gesamten für die RTA nötigen Code (RTA code) und speziell die Zeit nur für den Analyse- und Optimierungslauf der RTA (RTA run) gemessen. Zusätzlich wurde zum Vergleich die zugehörige reale Laufzeit des Compilers angegeben.

Die Ergebnisse sind in Tabelle 4.5 dargestellt. Die Werte sind Mittelwerte mehrerer Messungen. Die einzelnen Messwerte sind in Tabellen im Anhang in Abschnitt A.1 angegeben mitsamt einer kurzen statistischen Betrachtung und weiteren Details.

program	RTA code	RTA run	compiler
NQueensArray	104	12	107748
Multigrid	176	69	151524

Tabelle 4.5.: RTA-Laufzeiten während der X10i-Compilerlaufzeit (in Millisekunden)

Auf den ersten Blick fällt auf wie gering die Laufzeit der RTA gegenüber der Gesamtlaufzeit des Compilers ist. Die Verlängerung der Compilerlaufzeit durch die RTA erscheint anhand dieser Zahlen unbedeutend. Daneben ist interessant, dass die Gesamtzeit des für die RTA nötigen Codes gegenüber der des eigentlichen RTA-Laufs einen Overhead enthält, der auf eine aufwendig Suche nach den benötigten Entities und Typen für die Initialisierung der RTA zurückzuführen ist.

5. Fazit und Ausblick

5.1. Zusammenfassung

In dieser Arbeit wurde eine funktionsfähige Implementierung der Rapid Type Analysis (RTA) für Firm implementiert um dynamisch gebundene Methodenaufrufe zu devirtualisieren.

Es wurde erklärt, wie die RTA auf der Zwischendarstellung Firm durchgeführt werden kann und welche Besonderheiten dabei zu beachten sind. Für die Erkennung der Objekterzeugung, welche in Firm nicht klassisch anhand von Konstruktoraufrufen gemacht werden kann, wurde mit dem neuen Firm-Knoten `VptrIsSet` eine gute Lösung gefunden.

Diese Arbeit ist auch eine Art Fallstudie, wie sich die RTA fast 20 Jahre nach ihrer Entwicklung einsetzen lässt und welche Probleme dabei auftreten. Es stellte sich heraus, dass die Einhaltung der Annahmen, die die RTA aufstellt (vollständige Klassenhierarchie und Wissen über alle Objekterzeugungen), in der Praxis schwierig zu garantieren ist. Insbesondere ist es dem Analysecode nicht möglich selbstständig für Korrektheit zu sorgen. Da auch dynamisches Nachladen von Klassen und generische Objekterzeugung mittels Reflection nicht unterstützt werden können, ist die Verwendung der RTA mit modernen Programmiersprachen (wie z.B. Java) eingeschränkt. Zusätzlich ist die Unterstützung von nativen Methoden im zu übersetzenden Programm selbst ein offenes Problem.

In Kapitel 4 wurde durch Messungen festgestellt, dass die RTA-Implementierung dieser Arbeit durchaus Wirkung zeigen kann, wenn auch, wie typisch für die RTA, erst so richtig im Zusammenspiel mit einer Inlining-Optimierung. Wie von der RTA erwartet, erhöht die RTA-Implementierung die Compilerlaufzeiten höchstens in geringem Maße. Zum Teil konnten sogar Verringerungen beobachtete werden, die vermutlich darauf zurückzuführen sind, dass das Backend durch die Devirtualisierungen einige Programmteile weniger bearbeiten muss. Dies könnte durch explizites Entfernen nicht-lebendiger Funktionen und Klassen noch verbessert werden.

5.2. Zukünftige Arbeiten

Ein nächster Schritt wäre der Versuch, die Analyse zu verbessern, z.B. durch die von Namolaru [4] beschriebene Erweiterung um Datenflussanalyse oder durch Entwicklung einer ähnlichen interprozeduralen Type-Propagation-Analyse. Die RTA ist absichtlich einfach gehalten um nicht zu aufwendig zu werden und die Compilerlaufzeiten nicht zu stark anwachsen zu lassen. Dadurch lässt sie aber viel Potential für Verbesserung. Die

RTA betrachtet nicht, dass die Nutzung von Objekten bestimmter Klassen gegebenenfalls auf voneinander getrennte Teile des Programms beschränkt sein kann. Diese Lücke könnte eine Type-Propagation-Analyse vielleicht schließen ohne zu aufwendig zu werden.

Daneben kann die RTA um weitere Optimierungen ergänzt werden, die sie anhand der erstellten Lebendigkeitsinformationen durchführen kann. Das Entfernen von ungenutzten Methoden und Klassen war in dieser Arbeit geplant und angefangen, konnte aber nicht mehr rechtzeitig fertiggestellt werden. Darüberhinaus sind gegebenenfalls auch Optimierungen von Typvergleichen, Typprüfungen (vgl. `instanceof`) und Type-Casts möglich [2].

Da eine C++-Testumgebung für die vorliegende Arbeit nicht zur Verfügung stand, war es nicht möglich an der vollen Unterstützung von C++-Code zu arbeiten. Die Unterstützung von Funktionszeigern ist unter Umständen nicht vollständig und in C++ gibt es zusätzlich spezielle Member-Funktionszeiger. Außerdem hat C++ eine besondere Semantik für dynamisch gebundene Aufrufe innerhalb von Konstruktoren. Bei C++ ist auch zu beachten, dass es Funktionen außerhalb von Klassen aus Bibliotheken heraus geben kann, die als solche erkannt und gegebenenfalls behandelt werden müssen. Darüberhinaus kann die RTA unter Umständen virtuelle Vererbung optimieren (Devirtualization of Inheritance) [2].

Literaturverzeichnis

[1] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA'96*, 1996.

[2] David F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California at Berkley, 1998.

[3] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the Ninth European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[4] Mircea Namolaru. Devirtualization in GCC. In *GCC Developers' Summit 2006*, pages 125–133. IBM Haifa Research Lab, 2006.

[5] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. Firm—A Graph-Based Intermediate Representation. Technical Report 35, Karlsruhe Institute of Technology, 2011.

[6] libFirm. `http://pp.ipd.kit.edu/firm` .

[7] C. Click and M. Paleczny. A Simple Graph-Based Intermediate Representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 35–49, 1995.

[8] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27. ACM, 1988.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[10] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and Efficient Construction of Static Single Assignment Form. In Ranjit Jhala and Koen Bosschere, editors, *Compiler Construction*, volume 7791 of *Lecture Notes in Computer Science*, pages 102–122. Springer Berlin Heidelberg, 2013.

[11] X10 Language. `http://x10-lang.org` .

[12] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. An X10 Compiler for Invasive Architectures. Technical Report 9, Karlsruhe Institute of Technology, 2012.

[13] DFG Sonderforschungsbereich 89 – Invasive Computing (InvasIC). `http://pp.info.uni-karlsruhe.de/projects/invasic/invasic.php` `http://www.invasic.de` .

[14] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, chapter Invasive Computing: An Overview, pages 241–268. Springer, Berlin, Heidelberg, 2011.

[15] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152, 2003.

[16] GNU Compiler for the Java Programming Language (GCJ). `http://gcc.gnu.org/java` .

[17] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[18] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[19] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 108–124. ACM, 2001.

[20] n-queens problem. `http://en.wikipedia.org/wiki/Eight_queens_puzzle` .

[21] Hans-Joachim Bungartz, Christoph Riesinger, Martin Schreiber, Gregor Snelting, and Andreas Zwinkau. Invasive Computing in HPC with X10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, X10 '13, pages 12–19, New York, NY, USA, 2013. ACM.

[22] Mehrgitterverfahren. `http://de.wikipedia.org/wiki/Mehrgitterverfahren` .

Erklärung

Hiermit erkläre ich, Steffen Knoth, dass ich die vorliegende Studienarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

A. Anhang

A.1. Messungen

In diesem Abschnitt befinden sich die Tabellen mit den Werten der Einzelmessungen zu jedem Testprogramm.

In jeder Tabelle wird zusätzlich eine kurze statistische Auswertung gemacht. Es ist der Mittelwert (average), die Standardabweichung (stddev) und das Ergebnis eines zweiseitigen Student-t-Tests unter Annahme gleicher Varianz (p-value) angegeben. Diese Werte wurden mithilfe von Libreoffice 4.3.6 bestimmt.

Die Laufzeitmessungen wurden mit dem Unixkommando `time` durchgeführt. Von den Werten, die `time` ausgibt, wurde nur die Userland-Zeit genommen mit der Begründung, dass nur auf diese der Compiler einen Einfluss hat. Zudem sind alle Testprogramme der Natur, dass sie nur Berechnungen ausführen und das Betriebssystem kaum in Anspruch nehmen. Eine Ausnahme ist die Messung der RTA-Phase, die mittels `System.currentTimeMillis` erfolgte und deshalb Realzeit angibt. Für die Werte des gesamten Compilerlaufs wurde bei dieser Messung dann auch die Realzeit von `time` genommen.

LoopDynCall	default	RTA	RTA+Inlining	only Inlining
	6,73	6,76	1,11	2,65
	6,73	6,75	1,10	2,66
	6,76	6,76	1,11	2,67
	6,73	6,74	1,10	2,68
	6,74	6,75	1,07	2,65
	6,74	6,73	1,10	2,69
	6,71	6,77	1,11	2,66
	6,73	6,74	1,09	2,66
	6,76	6,76	1,10	2,67
	6,73	6,75	1,12	2,68
average	6,736	6,751	1,101	2,667
stddev	0,015	0,012	0,014	0,013
p-value default : x	-	0,024	<0,001	<0,001
p-value RTA : RTA+Inlining	-	-	<0,001	-
p-value RTA+Inling : Inlining	-	-	-	<0,001

Tabelle A.1.: einzelne Messergebnisse für LoopDynCall (in Sekunden)

LoopInterfaceCall	default	RTA	RTA+Inlining	only Inlining
	38,72	6,74	1,09	35,38
	38,72	6,74	1,09	35,39
	38,70	6,72	1,10	35,37
	38,71	6,72	1,11	35,37
	38,74	6,72	1,12	35,36
	38,69	6,73	1,08	35,38
	38,71	6,75	1,10	35,38
	38,71	6,74	1,09	35,36
	38,70	6,73	1,11	35,36
	38,72	6,72	1,10	35,37
average	38,712	6,731	1,099	35,372
stddev	0,014	0,011	0,012	0,010
p-value default : x	-	<0,001	<0,001	<0,001
p-value RTA : RTA+Inlining	-	-	<0,001	-
p-value RTA+Inling : Inlining	-	-	-	<0,001

Tabelle A.2.: einzelne Messergebnisse für LoopInterfaceCall (in Sekunden)

NQueensArray	default	RTA	RTA+Inlining	only Inlining
	5,21	5,19	1,84	1,86
	5,18	5,17	1,87	1,86
	5,21	5,20	1,85	1,90
	5,18	5,21	1,85	1,86
	5,20	5,18	1,89	1,88
	5,19	5,17	1,85	1,86
	5,18	5,19	1,85	1,86
	5,20	5,22	1,87	1,91
	5,19	5,17	1,87	1,86
	5,17	5,18	1,87	1,88
average	5,191	5,188	1,861	1,873
stddev	0,014	0,018	0,015	0,019
p-value default : x	-	0,67	<0,001	<0,001
p-value RTA : RTA+Inlining	-	-	<0,001	-
p-value RTA+Inling : Inlining	-	-	-	0,135

Tabelle A.3.: einzelne Messergebnisse für NQueensArray (in Sekunden)

NQueensArrayList	default	RTA	RTA+Inlining	only Inlining
	23,32	7,51	2,04	18,42
	23,33	7,49	2,04	18,42
	23,31	7,50	2,03	18,43
	23,34	7,49	2,05	18,41
	23,34	7,50	2,03	18,46
	23,35	7,50	2,06	18,44
	23,33	7,50	2,03	18,44
	23,32	7,50	2,02	18,43
	23,35	7,50	2,06	18,44
	23,35	7,50	2,03	18,42
average	23,334	7,499	2,039	18,431
stddev	0,014	0,006	0,014	0,014
p-value default : x	-	<0,001	<0,001	<0,001
p-value RTA : RTA+Inlining	-	-	<0,001	-
p-value RTA+Inling : Inlining	-	-	-	<0,001

Tabelle A.4.: einzelne Messergebnisse für NQueensArrayList (in Sekunden)

Multigrid	default	RTA	RTA+Inlining	only Inlining
	31,69	34,14	27,18	30,96
	32,51	33,64	26,04	30,20
	32,32	34,55	26,57	31,25
	32,45	34,49	26,38	29,18
	32,44	34,59	25,50	30,66
	32,89	34,11	26,18	29,82
	32,03	33,97	26,37	30,20
	32,01	34,38	25,95	29,10
	32,36	34,49	26,57	30,70
	31,81	34,71	26,01	31,24
average	32,251	34,307	26,275	30,331
stddev	0,363	0,333	0,454	0,778
p-value default : x	-	<0,001	<0,001	<0,001
p-value RTA : RTA+Inlining	-	-	<0,001	-
p-value RTA+Inling : Inlining	-	-	-	<0,001

Tabelle A.5.: einzelne Messergebnisse für Multigrid (in Sekunden)

```
class A {
    public def f(x:int, y:int):int {
        return x+y;
    }
}

public class LoopDynCall {
    static def calc() {
        val a = new A();
        var sum:int = 0;
        for (var i:int=0; i<=1000000000; i++) {
            sum = a.f(sum, i);
        }
        return sum;
    }
    public static def main(Array[String]) {
        calc();
    }
}
```

Listing A.1: Quellcode von LoopDynCall

```
interface Iface {
    public def f(x:int, y:int):int;
}

class A implements Iface {
    public def f(x:int, y:int):int {
        return x+y;
    }
}

public class LoopInterfaceCall {
    static def calc() {
        val a:Iface = new A();
        var sum:int = 0;
        for (var i:int=0; i<=1000000000; i++) {
            sum = a.f(sum, i);
        }
        return sum;
    }
    public static def main(Array[String]) {
        calc();
    }
}
```

Listing A.2: Quellcode von LoopInterfaceCall

X10i Compilerlaufzeit Multigrid	default	RTA
	208,12	202,73
	212,56	191,85
	204,73	197,39
	205,55	199,87
	202,29	197,53
average	206,65	197,87
stddev	3,905	4,006
p-value	-	0,008

Tabelle A.6.: einzelne Messergebnisse für das Kompilieren von Multigrid (in Sekunden)

X10i Compilerlaufzeit NQueensArray	default	RTA
	135,82	140,98
	141,02	139,89
	144,69	143,62
	138,90	144,86
	143,97	141,29
average	140,88	142,13
stddev	3,661	2,044
p-value	-	0,524

Tabelle A.7.: einzelne Messergebnisse für das Kompilieren von NQueensArray (in Sekunden)

RTA-Laufzeit Multigrid	RTA code	RTA run	compiler
	174	71	151261
	180	69	153398
	175	69	150789
	176	68	150647
	181	72	152373
average	177,2	69,8	151694
stddev	3,114	1,643	1169

Tabelle A.8.: einzelne Messergebnisse der RTA-Laufzeit beim Kompilieren von Multigrid (in Millisekunden)

RTA-Laufzeit NQueensArray	RTA code	RTA run	compiler
	90	12	107203
	99	12	105259
	97	12	107476
	106	12	107625
	126	13	111175
average	103,6	12,2	107748
stddev	13,759	0,447	2140

Tabelle A.9.: einzelne Messergebnisse der RTA-Laufzeit beim Kompilieren von NQueensArray (in Millisekunden)