Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

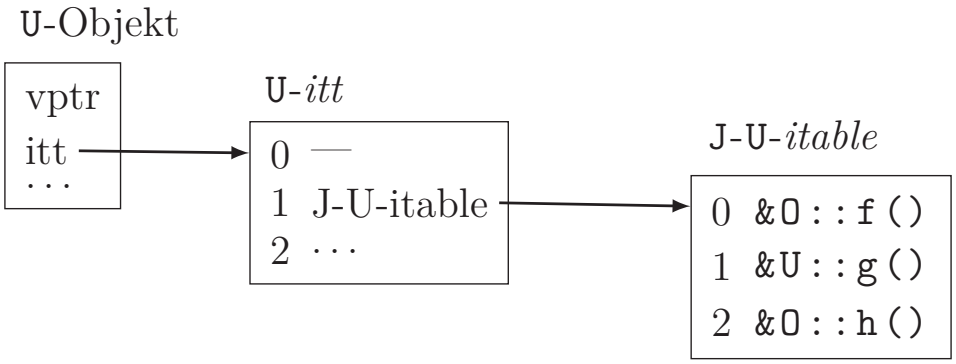
Lehrstuhl Prof. Dr.-Ing. Snelting

Verbesserte Implementierung von Interface-Aufrufen

Bachelorarbeit von

Philipp Serrer

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. Jörg Henkel
Betreuende Mitarbeiter:	Dipl.-Inform. Manuel Mohr

Bearbeitungszeit: 03. August 2015 – 01. Dezember 2015

Zusammenfassung

Viele objektorientierte Sprachen unterstützen für Klassen nur Einfachvererbung, bei Interfaces aber Mehrfachvererbung. Prinzipiell ist es möglich den *vtable*-Mechanismus für dynamisch gebundene Methoden zu verwenden. Dies würde aber größere Objekte und erhöhte Komplexität mit sich führen. libFirm bzw. liboo verwendet für Interface-Aufrufe aktuell eine Laufzeitsuche, welche sehr langsam ist. Aufgrund der Notwendigkeit diese zu beschleunigen, werden in dieser Arbeit mehrere effiziente tabellenbasierte Methoden vorgestellt um Interface-Aufrufe auszuführen. Dabei werden die Methoden *Searched ITables* und *Directly Indexed ITables* implementiert und evaluiert.

Inhaltsverzeichnis

1. Einführung **7**

2. Grundlagen und Verwandte Arbeiten **9**

2.1. Dynamische Bindung . . . 9
2.2. *vtable*-Mechanismus . . . 10
2.2.1. Aufbau der *vtables* . . . 11
2.2.2. Aufruf virtueller Methoden . . . 11
2.3. Mehrfachvererbung . . . 12
2.3.1. Interfaces und Mehrfachvererbung . . . 13
2.4. Interface-Methodenaufrufe . . . 14
2.4.1. Class Object Search . . . 14
2.5. Interface-Methodenaufrufe mit *itables* . . . 15
2.5.1. Directly Indexed ITables . . . 16
2.5.2. Searched ITables . . . 17
2.6. Bidirektionale Vertikale Methodentabellen . . . 18
2.7. Firm . . . 20
2.7.1. libFirm, liboo . . . 21
2.8. X10i . . . 22
2.9. Bytecode2firm . . . 22

3. Entwurf und Implementierung **23**

3.1. Ablauf der Implementierung . . . 23
3.2. Rekursive Klassenhierarchien . . . 24
3.2.1. Klassen Iterator . . . 24
3.3. Implementierung von *Directly Indexed ITables* . . . 25
3.3.1. Erstellung der *itable* . . . 25
3.3.2. Erstellung der *itt*'s . . . 26
3.3.3. Ersetzung des MethodSel-Knoten . . . 26
3.4. Implementierung von *Searched ITables* . . . 27
3.4.1. Erstellung der *itable* . . . 27
3.4.2. Erstellung der *itt* . . . 27
3.4.3. Ersetzung des MethodSel-Knoten . . . 28
3.5. Bidirektionale Vertikale Methodentabelle . . . 29

4. Evaluation **31**
4.1. Anmerkung zur Ausführung/Auswertung der Testfälle 31
4.2. Synthetische Testfälle 32
4.3. Real-World Test 34
4.4. Getrennte Übersetzungseinheiten 35

5. Fazit und Ausblick **37**
5.1. Zusammenfassung 37
5.2. Zukünftige Arbeiten 37

A. Anhang **43**
A.1. Code 43

1. Einführung

Interfaces sind ein integraler Bestandteil moderner Software Architekturen. Mit ihrer Hilfe kann man weitere Abstraktionsebenen anlegen, ohne an die Vererbungshierarchie gebunden zu sein. Es wird lediglich ein Vertrag definiert, der von allen implementierenden Klassen erfüllt werden muss. Dabei bleiben alle Vorteile von Polymorphismus erhalten. Durch die Vorteile, die sich daraus ergeben, arbeitet man oft nicht mehr mit konkreten Klassen, sondern verwendet fast ausschließlich Interfaces. Ebenfalls bieten Interfaces eine einfache Form der Mehrfachvererbung.

Betrachtet man die Standard Bibliotheken von X10 oder Java, so implementieren viele Klassen eine Vielzahl an Interfaces. Bei ArrayList aus X10 sind es aktuell acht Interfaces, bei Java sechs. Business-Frameworks enthalten teilweise Klassen mit mehr als 20 Interfaces. Der *vtable*-Mechanismus für Mehrfachvererbung eignet sich nicht, da Objekte häufig deutlich größer würden und die Komplexität erheblich steigt. Bisher wurde eine einfache Laufzeitsuche nach der konkreten Funktion aufgerufen. Dieser Ansatz ist allerdings inhärent langsamer als dynamisch gebundene Methodenaufrufe für Objekte.

Wie kann man Methodenaufrufe auf Interface-Typen effizient ausführen?

Dabei bieten sich mehrere Möglichkeiten an. In dieser Arbeit werden *Directly Indexed ITables* und *Searched ITables* implementiert. Beides sind tabellenbasierte Ansätze, die sich hauptsächlich dadurch unterscheiden, wie die richtige Methodentabelle gefunden wird. Dies bestimmt auch, wie gut das Laufzeitverhalten ist, bzw. wie flexibel man diese Methode einsetzen kann. So ist zusätzlich zu klären, welche Ansätze mit getrennten Übersetzungseinheiten umgehen können, oder wie hoch der hierfür nötige Speicherbedarf ist.

Im Kapitel 2 werden Grundlagen und verwandte Arbeiten vorgestellt. Kapitel 3 beschäftigt sich mit der Implementierung von *Searched ITables* und *Directly Indexed ITables*. Das Ergebnis der Arbeit wird in Kapitel 4 ausgewertet und mit dem aktuellen Stand verglichen. Abschließend wird in Kapitel 5 ein Fazit gezogen und aufgeführt, wie zukünftige Arbeiten darauf aufbauen können.

2. Grundlagen und Verwandte Arbeiten

Es werden zunächst grundlegende Konzepte und Techniken vorgestellt, mit denen ein Interface-Methodenaufruf ausgeführt wird.

2.1. Dynamische Bindung

Eine inhärente Eigenschaft von Objektorientierung ist die Verwendung von dynamischer Bindung bzw. polymorpher Methoden.

Deklariert man eine Variable `x` vom Typ `O` (Siehe Abb 2.1), hat diese den statischen Typ `O`. Sie kann alle Objekte vom Typ `O` referenzieren. Dazu gehören auch alle Subtypen, d.h. Typen, die von `O` erben - wie z.B. `U1` oder `U2`.

Ruft man nun eine Methode `f()` auf `x` auf, so ist nicht klar, welchen Typ `x` referenziert. Im Falle von dynamischer Bindung wird für den Methodenaufruf `f()` nicht der statische Typ `O` von `x` verwendet, sondern zur Laufzeit der dynamische Typ ermittelt und auf diesem `f()` ausgeführt.

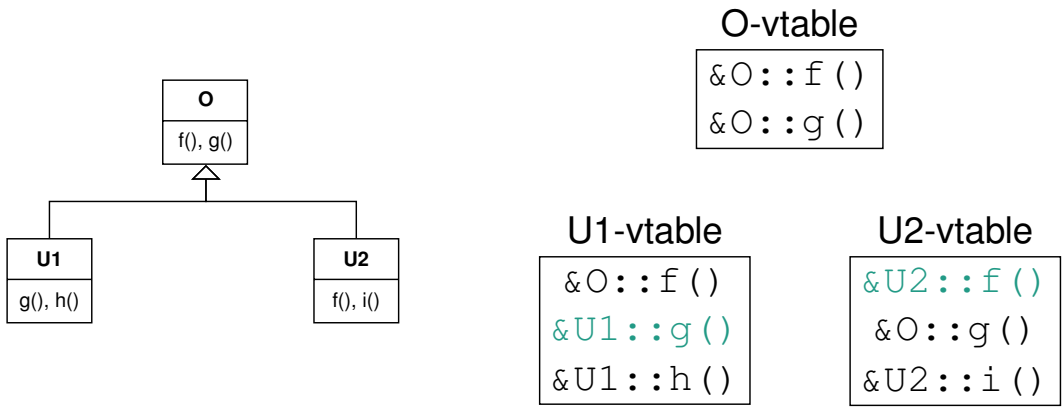


Abbildung 2.1.: Klassendiagramm

Wird betrachten nun folgenden Code

```
((O)(new U1())).g();
```

Hier wird ein neues `U1`-Objekt erstellt, ein Upcast auf `O` ausgeführt und auf `O` dann `g()` aufgerufen. Da `g()` dynamisch gebunden ist, wird hier aber nicht `O::g()` ausgeführt, sondern `U1::g()`.

Im Allgemeinen kennt der Compiler zur Kompilationszeit noch nicht das Aufrufziel, da dieses zum Beispiel von Benutzereingaben abhängt. Die konkrete Methode muss zur Laufzeit ermittelt werden.

Mit Hilfe von statischer Codeanalyse kann der Compiler in Ausnahmefällen schon beim Kompilieren feststellen, welchen Typ das Objekt hat und die Methode statisch binden. Die dynamische Bindung entfällt dann und es reicht ein einfacher `CALL`-Maschinenbefehl um die Methode auszuführen. Beliebete Analysemethoden sind Rapid-Type Analysis[3] (RTA) oder Points-to Analysis [2] [12].

Eine effiziente Methode um dynamische Bindung zu realisieren sind *vtables*, welche in folgendem Abschnitt 2.2 vorgestellt werden.

2.2. *vtable*-Mechanismus

Damit sich dynamische Bindung etablieren konnte, brauchte es eine effiziente Umsetzung. Eine Laufzeitsuche für jeden virtuellen Funktionsaufruf ist aus performancetechnischen Gründen nicht tragbar und hätte dafür gesorgt, dass sich Polymorphismus in der Objektorientierung nie durchgesetzt hätte.

Der *vtable*-Mechanismus bietet für dieses Problem eine sehr schnelle und einfache Lösung. Mit Hilfe einer vorberechneten Tabelle kann man den Funktionsaufruf auf nur drei Maschinenbefehle kürzen.

Die Idee hinter den *vtables* besteht darin, dass man zu jeder Klasse eine Tabelle anlegt, in der die statischen Adressen der Funktionen (Funktionszeiger) aller dynamisch gebundener Methoden aufgelistet sind. In einer Vererbungshierarchie befinden sich alle Methoden mit gleicher Signatur an demselben Index. Da der Index der Methode schon zur Kompilationszeit bekannt ist, kann man den Funktionsaufruf zur Laufzeit auf zwei Ladebefehle und einen Funktionsaufruf verkürzen.

Ein Beispiel für *vtables* findet sich in Abbildung 2.1. Man erkennt hier, wie die *vtable* von `O` an Index 0 einen Zeiger auf die Funktion `O::f()` hat. Da die Klasse `U2` die Methode

`O::f()` überschreibt, wird auch der Eintrag durch `U::f()` ersetzt. In der Klasse `U1` hingegen bleibt `O::f()` unverändert.

2.2.1. Aufbau der *vtables*

Pro Klasse existiert eine *vtable*. In dieser stehen alle dynamisch gebundenen Methoden, welche auf der Klasse aufgerufen werden können. Falls die Klasse eine Elternklasse hat, erbt diese die Struktur der *vtable* der Elternklasse.

Methoden, die in der Klasse überschrieben werden, werden auch in der *vtable* überschrieben (Siehe `U1::g()` in der `U1`-*vtable* aus Abb 2.1). Neue Methoden werden an das Ende der Tabelle angehängt (Siehe `U1::h()`). Dadurch wird sichergestellt, dass sich eine Methode `foo()` von `A` auch in allen Klassen, die von `A` erben, an derselben Stelle befindet. Die Reihenfolge der neu hinzugefügten Methoden entspricht in der Regel der Reihenfolge, wie sie in der Klasse definiert wurden - dies ist aber nicht notwendig.

Durch diesen Aufbau ist sichergestellt, dass sich in einer Vererbungshierarchie jede Methode immer an derselben Stelle befindet. Die *vtable* wird an einer festen Adresse gespeichert, deren Adresse als sogenannter *vptr* im Objektlayout abgelegt wird.

2.2.2. Aufruf virtueller Methoden

Um eine virtuelle Methoden mit Hilfe von *vtables* aufzurufen, folgt man dem *vptr* und lädt innerhalb der *vtable* den Eintrag für die aufzurufende Methode, welcher die Funktionsadresse enthält. Diese ruft man mit einem einfachen `CALL`-Befehl auf. Alle hierfür notigen Schritte haben unabhängig von der Methodenanzahl oder Tiefe der Vererbungshierarchie konstanten Aufwand und somit auch der komplette Aufruf virtueller Methoden.

Da der *vptr* immer an der selben Stelle am Anfang des Objektlayouts gespeichert wird(`*(x + offset(vptr))`), findet man die *vtable* unabhängig von Typ des Objekts - insbesondere auch nach Up- bzw. Down-Casts.

Folgender C-Code [9, Folie 30] verdeutlicht den Aufruf einer dynamisch gebundenen Methode.

Listing 2.1: Dynamische Bindung, C

```
1 (*(*(x + offset(vptr)) + offset(methode)))()
```

Listing 2.2: Dynamische Bindung, ASM

```
1 load [x + OFFSET_VPTR], reg0
2 load [reg0 + OFFSET_methode], reg1
3 CALL reg1
```

2.3. Mehrfachvererbung

Bei der Mehrfachvererbung hat eine Klasse mehrere direkte Oberklassen. So erbt eine Klasse `HiWi` beispielsweise gleichzeitig von `Student` und `Angestellter`. Da aber ein Student nicht zwangsläufig ein Angestellter ist und auch nicht jeder Angestellte studiert, muss die Klasse `HiWi` gleichzeitig von beiden Klassen erben.

Die Implementierung von Mehrfachvererbung ist komplex, weshalb nur wenige Sprachen (zum Beispiel C++) diese Funktionalität unterstützen.

Die Probleme, die sich hierbei ergeben, sind unter anderem:

- Mehrdeutigkeiten:
  `Student` und `Angestellter` implementieren jeweils die Methode `schlafen()`. `HiWi` überschreibt `schlafen()` aber nicht. Was wird nun bei `HiWi::schlafen()` aufgerufen?

- Erweiterung des Objektlayouts:
  In jedem Objekt muss pro Elternklasse ein eigener *vptr* gespeichert werden, der auf die passende *vtable* zeigt.

- Anpassung der *vtable*:
  Zusätzlich zu den Funktionszeigern muss die *vtable* um Deltas ergänzt werden, die man für dynamisch gebundene Funktionsaufrufe benötigt.

- Aufwändigere Funktionsaufrufe:
  Im Allgemeinen sind Funktionsaufrufe bei Mehrfachvererbung teurer als für Einfachvererbung [13].

  ```
  1   // u->f();
  2   register vt = &(u->vptr[0]);
  3   *(vt->fct)(u + (vt->delta));
  ```

  Durch die Verwendung von Thunks [14] lässt sich das teilweise beheben, dies verkompliziert die Logik aber.

- Casts sind nicht mehr trivial:
  Da eine Klasse mehrfach in der Objekthierarchie vorhanden sein kann, können Casts mehrdeutig werden. Zudem muss der Objektzeiger ggf. verschoben werden.

- Vergleich von Referenzen erfordert mehr Logik:
  Das gleiche Problem wie bei den Casts, ergibt sich auch beim Vergleich von Referenzen verschiedener Typen, da durch den Casts der Objektzeiger ggf. verschoben wurde. So ist z.B. `Foo f = new Foo(); Bar b = f;` aber `f` und `b` zeigen dennoch auf verschiedene Speicherstellen.

- Null-Zeiger:
  Da der Null-Zeiger auf kein Objekt zeigt, müssen ggf. auch noch Null-Zeiger-Tests ergänzt werden.

2.3.1. Interfaces und Mehrfachvererbung

Jedes Interface lässt sich auch als abstrakte Klasse schreiben, die nur abstrakte Methoden enthält. Man kann also auch Interface-Methodenaufrufe dadurch auflösen, indem man jedes Interface in eine abstrakte Klasse transformiert und für Interface-Aufrufe den *vtable*-Mechanismus von Mehrfachvererbung verwendet.

Dadurch hätte man für alle Interface-Aufrufe eine effiziente Implementierung.

Bei näherer Betrachtung dieser Idee zeigen sich aber konzeptionelle Schwächen, die diese Idee ausscheiden lassen. Das Hauptproblem ist die enorme Komplexität, die man sich dadurch einhandelt und vermeiden möchte.

- Die Implementierung ist komplex und aufwändig:
  Der bereits bestehende *vtable*-Mechanismus müsste komplett überarbeitet werden. Zusätzlich muss man Casts und Referenz-Vergleiche erweitern.

- Das Objektlayout wird größer:
  Für jede Oberklasse muss ein zusätzlicher *vptr* im Objekt gespeichert werden.

  Wie in der Einleitung erwähnt, werden Interfaces sehr häufig und oft eingesetzt. Alleine in der Standard-Bibliothek von Java gibt es Klassen mit 6 und mehr Interfaces. In Business Bibliotheken ist dies teilweise noch viel dramatischer. Als Beispiel sei hier die Klasse `GroovyWebApplicationContext` [11] des *Spring*-Frameworks erwähnt. Diese implementiert aktuell 21 Interfaces. Ohne Felder würde jedes dieser Objekte auf einer 64-bit-Architektur bereits mindestens 168 Bytes an Speicher belegen.

2.4. Interface-Methodenaufrufe

Es werden nun verschiedene Verfahren vorgestellt, mit denen man Interface-Methodenaufrufe ausführen kann. Jede hiervon hat ihre eigenen Vor- und Nachteile, weshalb die geeignete vom Anwendungsfall abhängt.

2.4.1. Class Object Search

Class Object Search [1] ist ein naiver Ansatz für Interface Methodenaufrufe. Die Grundidee dahinter ist, dass zur Laufzeit die passende Methode gesucht und ausgeführt wird.

Es werden die Laufzeit-Typ-Informationen (RTTI) aus dem Objekt geladen und mit Hilfe dieser die Klassenhierarchie von unten nach oben nach der passenden Methode durchsucht. Sobald eine Methode mit passender Methodensignatur gefunden wird, wird diese zurückgegeben und ausgeführt. Zur Methodensignatur gehören hierbei Name, Parameter und Rückgabetyp.

Input : klass; name; signature
Output : method
$k \leftarrow$ klass
repeat
  foreach *method* $\in$ *k.methods* **do**
    if *method.name* = *name AND method.signature* = *signature* **then**
      return method.funcptr
  $k \leftarrow$ *klass.parent*
until $k \neq NULL$

Abbildung 2.2.: Class Object Search: *klass* enthält die RTTI (Methoden, Klassenhierarchie, ...), *name* den Namen der gesuchten Methode, *signature* die Signatur. Diese Methode wird zur Laufzeit aufgerufen, um die passende Methode zu einem Interface-Aufruf zu finden.

Bei dieser Implementierung des Interface Lookup fallen drei Punkte auf, welche die Performance negativ beeinflussen.

- Die Tiefe der Vererbungshierarchie:
  Im schlimmsten Fall müssen alle n Elternklassen des Objekts durchsucht werden. Dies wirkt sich bei tiefen Vererbungshierarchien, in denen eine Methode weit oben (z.B. der Root-Klasse) definiert wurde, negativ aus.

- Die Menge der Methoden:
  Für jede betrachtete Klasse müssen bis zu einem Treffer potentiell alle m Methoden betrachtet werden. Jede später neu hinzugefügte Methode wirkt sich daher auch negativ auf die Laufzeit für Interface-Methodenaufrufe bereits vorhandener Methoden aus.

- Die Länge der Methodensignatur:
  Es wird bei jeder Methode die Signatur verglichen. Da die Signatur als Zeichenkette der Länge k vorliegt, die aus Methodenname und Parametern besteht, wirken sich lange Methodennamen oder viele Parameter negativ aus.

Zusammengefasst lässt sich über die Laufzeit der Laufzeitsuche sagen: $\mathcal{O}(n \cdot m \cdot k)$.

Im Folgenden Abschnitt werden tabellenbasierte Techniken vorgestellt, die effizienter sind.

2.5. Interface-Methodenaufrufe mit *itables*

Analog zur den Methodentabellen für virtuelle Methodenaufrufe kann man auch Tabellen für Interface Methodenaufrufe anlegen. Diese „Interface-Tabellen“ werden in folgendem als *itables* bezeichnet.

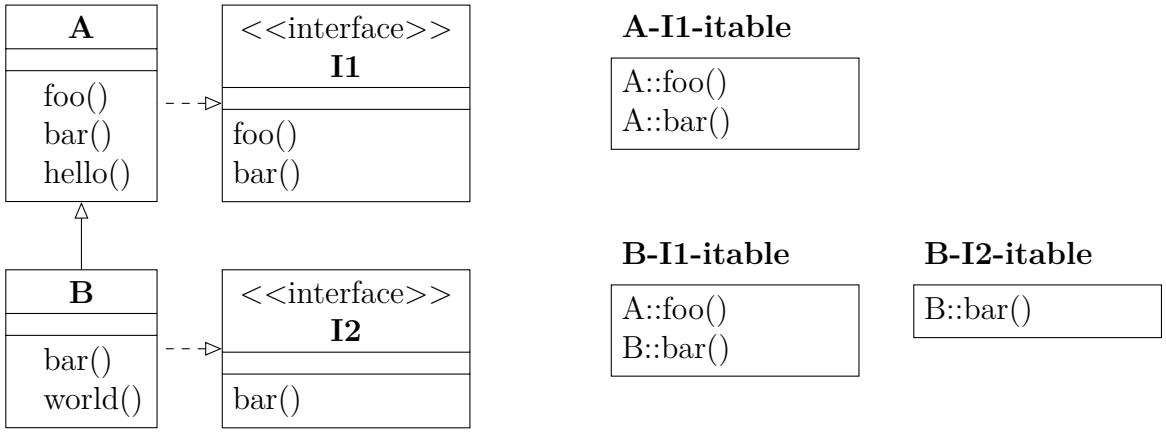


Abbildung 2.3.: *itables* Beispiel für die Klassen A und B

Eine *itable* bezieht sich immer auf ein Klassen-Interface-Tupel. In ihr werden alle Methoden der Klasse abgelegt, die in dem zugehörigen Interface deklariert wurden. Unabhängig von der zugehörigen Klasse, stehen in der *itable* die Methoden immer an einem festen Index pro *Interface*, der schon zur Kompilationszeit bekannt ist. Über die Adresse der *itable* und den Offset der Methode kommt man also an die Adresse der aufzurufenden Methode. Dies entspricht auch dem Methodenaufruf innerhalb der *vtable*.

```
1 (*(itable + offset(methode)))()
```

```
1 load [itable + OFFSET_methode], reg0
2 CALL reg0
```

Die Komplexität eines Interface-Methodenaufrufs reduziert sich dadurch auf das Auffinden der richtigen Interface-Tabelle.

2.5.1. Directly Indexed ITables

Bei *Directly Indexed ITables* (DIIT) trifft man die Annahme, dass zur Kompilationszeit bereits alle Interfaces und Klassen bekannt sind.

Man erstellt nun für jede relevante Klasse eine zusätzliche Tabelle, die wir zukünftig als *itt* (*itable*-Tabelle) bezeichnen. Diese hat so viele Einträge wie es Interfaces im Programm gibt. Für jedes Interface, das die Klasse implementiert, wird in der *itt* nun eine Referenz auf die passende *itable* gespeichert. Der Index, an dem eine Referenz auf die *itable* des Interface-*I* in der *itt* abgelegt wird, ist hierbei für jede *itt* identisch. Alle nicht benötigten Einträge in der *itt* verbleiben leer.

Zusätzlich wird das Objektlayout um ein weiteres Feld erweitert, das eine Referenz auf die *itt* speichert.

Listing 2.3: Code Beispiel für Abbildung 2.5

```
1  interface I { ... } // itt-index: 0
2  interface J {       // itt-index: 1
3          void f(); // itable-index: 0
4          void g(); // itable-index: 1
5          void h(); // itable-index: 2
6  }
7
8  class O {
9          void f() { ... }
10         void h() { ... }
11 }
12
13 class U extends O implements J {
14         void g() { ... }
15 }
```

Wird nun zur Laufzeit eine Interface-Methode aufgerufen, wird aus der Objekt-Referenz die *itt* geladen. Da jedes Interface einen eindeutigen Index bekommen hat, können wir

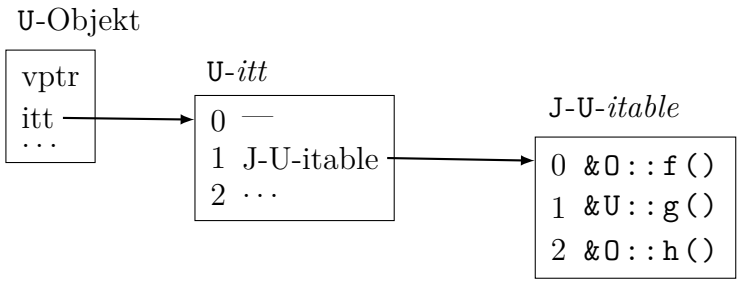


Abbildung 2.4.: ITables Schema für *Directly Indexed ITables* (Siehe Listing 2.3)

über diesen in der *itt* die Adresse der *itable* nachschlagen. Alles Weitere entspricht dem Aufruf einer Methode aus der *vtable*.

Wir benötigen nur eine Indirektion mehr als bei einem *vtable*-Methodenaufruf, da uns alle drei benötigten Offsets schon zur Kompilationszeit bekannt sind.

Wollen wir zum Beispiel `((J)new U).f()` aus Abbildung 2.5 aufrufen, so folgen wir dem *itt*-Zeiger, finden in der *itt* an Index 1 die Referenz auf die *J-U-itable* und in dieser die Methode `O::f()` an Index 0.

Listing 2.4: Directly Indexed ITables

```
1 load [x + OFFSET_itt], reg0
2 load [reg0 + OFFSET_interface], reg1
3 load [reg1 + OFFSET_methode], reg2
4 CALL reg2
```

2.5.2. Searched ITables

Auch bei *Searched ITables* (SIT) verwenden wir eine *itt*. Im Gegensatz zu direkt indizierten ITables werden in der *itt* dieses Mal nur die *itable*-Referenzen gespeichert, die auch wirklich vorhanden sind. Dadurch verlieren wir zur Kompilationszeit die Information darüber, an welcher Stelle in der *itt* die Referenz zu der richtigen *itable* liegt. Dies bringt uns aber den Vorteil, dass wir die Kompilation in verschiedene Übersetzungseinheiten (ÜE) aufteilen können, die unabhängig voneinander später vom Linker zusammengeführt werden.

Um die *itable* zu finden erweitern wir die *itt*-Einträge um einen pro Interface eindeutigen *Identifier* nach dem wir zur Laufzeit suchen können. Dies könnte beispielsweise der Name des Interface sein. In Kapitel 3.4.2 wird aber noch eine bessere Alternative vorgestellt, die deutlich performanter ist und keine Stringvergleiche benötigt.

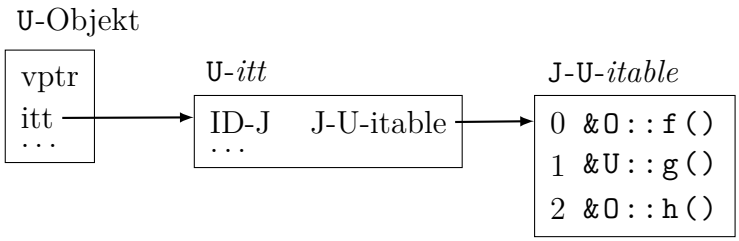


Abbildung 2.5.: ITables Schema für *Searched ITables* (Siehe Listing 2.3)

Wird nun zur Laufzeit eine Interface-Methode aufgerufen, so wird ein Laufzeitservice aufgerufen, der als Parameter das Objekt (bzw. gleich die *itt*) und den Identifier des Interfaces bekommt. Innerhalb der *itt* wird dann nach dem passenden Identifier gesucht und die dazugehörige *itable* zurück gegeben.

Das weitere Vorgehen ist wie bei den Directly Indexed ITables.

Nachfolgend (Listing 2.5) ist eine vereinfachte Darstellung der Searched ITables Methode dargestellt.

Listing 2.5: Vereinfachter Aufruf von Searched ITables

```
1 CALL oo_searched_itable_method(OBJ, ITABLE_ID, METHOD_
    INDEX_ITABLE), ret0
2 CALL ret0(...)
```

2.6. Bidirektionale Vertikale Methodentabellen

Bidirektionale Vertikale Methodentabellen wie in [18] beschrieben verfolgen einen anderen Ansatz ohne *itables*. Man trifft aber auch hier wieder die Annahme, dass zur Kompilationszeit bereits alle Klassen und Interfaces bekannt sind.

Es wird für jede relevante Klasse eine *Schnittstellentabelle* erstellt, die so wie die *vtable* Methodenzeiger auf die Interface-Methoden enthält.

Die Idee ist nun, dass über alle *Schnittstellentabellen* hinweg die Methodenzeiger an der selben Stelle stehen. D.h. wenn in Klasse `U` die Funktion `I::f()` an Index 42 steht, tut sie dies auch in Klasse `O`.

Damit dies funktioniert ordnet man jeder relevanten Methode aufsteigend eine Nummer zu, die als Index für die *Schnittstellentabelle* verwendet wird. Relevante Methoden sind alle, die irgendwo in einem Interface definiert wurden. Methodensignaturen die in mehreren Interfaces vorhanden sind, bekommen denselben Index. Dies ist möglich da

Interface-Methoden keine Implementierung im Interface besitzen und nur angeben, welche Methoden in einer Klasse vorhanden sein müssen.

Diese Tabelle kann man nun als Referenz in dem Objektlayout speichern, oder man verwendet den *vptr* für die virtuelle Methodentabelle mit. Es besteht hier die Möglichkeit, dass man die *vtable* in die negative Richtung wachsen lässt und alle Interface-Methoden mit negativem Index aufruft. Der *vptr* bleibt dabei unverändert auf dem ersten Element der *vtable*. Ein Beispiel findet sich in Abbildung 2.6.

Listing 2.6: Code für Abbildung 2.6

```
1  interface I {
2          void foo(); // INDEX -1
3  }
4  interface J {
5          void f();   // INDEX -2
6  }
7  interface K {
8          void g();   // INDEX -3
9          void h();   // INDEX -4
10 }
11
12 class O implements J {
13         void f() { ... }
14         void g() { ... }
15 }
```

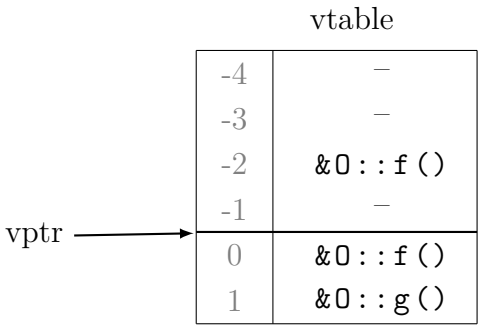


Abbildung 2.6.: Bidirektionale Vertikale Methodentabelle (Siehe Listing 2.6)

Der Aufwand für einen Interface-Aufruf ist mit dieser Methode identisch mit dem Aufruf dynamisch gebundener Methoden.

Damit diese Tabellen nicht unnötig groß werden, kann man alles nach dem letzten existierenden Eintrag in der *Schnittstellentabelle* abschneiden. Durch die statische Typprüfung ist sichergestellt, dass diese Methoden nie aufgerufen werden können. Allerdings bleibt

Listing 2.7: Interface-Methodenaufruf mit Hilfe von BVT. Siehe zum Vergleich Listing 2.2 für dynamisch gebundene Methodenaufrufe

```
1 load [x + OFFSET_VPTR], reg0
2 load [reg0 + OFFSET_INTERFACE_methode], reg1
3 CALL reg1
```

sehr viel freier Platz in der Tabelle, der sich nur schwer bis gar nicht weg optimieren lässt.

In Abbildung 2.6 ist eine Tabelle dargestellt, bei der man die obersten beiden Einträge (Index -4 und -3) entfernen kann. Da `O` das Interface `K` nicht implementiert, können auch niemals die Methoden von `K` auf `O` aufgerufen werden. Da diese Einträge nicht definiert sind, würde dies zu einem Fehler führen.

2.7. Firm

Die Umsetzung erfolgte als Teil von Firm innerhalb von liboo.

Firm wurde als Zwischendarstellung (IR) für den Sather-K Compiler Fiasco von Armbruster und von Roques [16] erstellt. Daher stammt auch der Name - Fiasco's Intermediate Representation Mesh.

Im Rahmen der Dissertation von Trapp [15] an der Universität Karlsruhe wurde Firm dann schrittweise erweitert und steht zwischenzeitlich als Open-Source-Bibliothek bereit.

Zwischenzeitlich steht Firm sowohl für die Zwischendarstellung als auch das Compilerframework libFirm. Die Zwischendarstellung ist Graphen-basiert und beruht auf der Arbeit von C. Click [6]. Der Graph liegt in der Static-Single-Assignment Form (SSA-Form) [10] vor, welche für viele Optimierungen vorteilhaft ist.

Die Idee hinter der SSA-Form ist, dass jeder Variable genau ein Wert zugewiesen wird. Dieser Wert wird im Firm-Graph dann als Abhängigkeitskante dargestellt. Auch viele andere populäre Compiler, wie die GNU Compiler Collection, verwenden aufgrund ihrer positiver Eigenschaften die SSA-Form [8].

2.7.1. libFirm, liboo

libFirm ist eine Implementierung von Firm und repräsentiert den Compiler-Unterbau. Mit Hilfe verschiedener Frontends werden aktuell unter anderem die Sprachen C99, Java oder X10 unterstützt. Es existieren aktuell ausgereifte Backends für die Architekturen IA32 und SPARC. MIPS, ARM und AMD64 werden noch nicht komplett unterstützt.

Da libFirm nur imperative Programmiersprachen unterstützt, wurde mit liboo eine Bibliothek entwickelt, die den Firm-Graph um eine weitere Ebene für Objektorientierung erweitert. Hierfür werden die bestehenden Knoten um weitere Attribute erweitert und neue Knoten, wie zum Beispiel der `MethodSel`-Knoten, hinzugefügt. Dadurch lassen sich beispielsweise Klassenhierarchien darstellen oder Methoden um Attribute wie *final*, *protected*, etc. erweitern.

An dieser Stelle setzt auch diese Bachelor-Arbeit an. Beim Aufruf von Methoden unterscheidet liboo zwischen *statischen*, *dynamisch gebundenen* und *Interface*-Methodenaufrufen. Hierbei wird der `MethodSel`-Knoten (siehe Abbildung 2.7) durch andere Knoten ersetzt, welche die passende Funktionsadresse zurück geben. Der `MethodSel`-Knoten beschreibt abstrakt die Auswahl des Aufrufziel, für ein Objekt und ist daher auch immer einem `CALL`-Knoten vorangestellt. Er ist eine Ergänzung von liboo um dynamische Bindung zu behandeln. Je nach Anwendungsfall wird der `MethodSel`-Knoten zum Beispiel in einen `Address`-Knoten oder weitere `CALL`-Knoten umgewandelt.

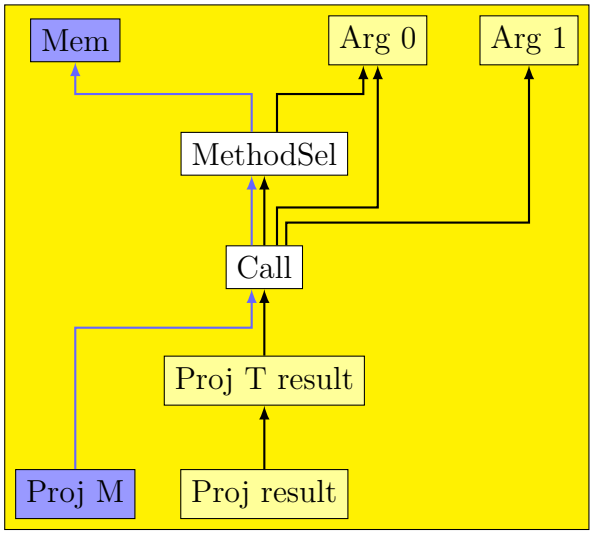


Abbildung 2.7.: Firm Graph mit MethodSel-Knoten

Bei *statischen* Methodenaufrufen wird einfach nur die Adresse der Methode aufgerufen. Das Backend generiert aus diesem Aufruf später einen einfachen `CALL` Befehl.

Dynamisch gebundene Methodenaufrufe verwenden den *vtable*-Mechanismus wie er in

Abschnitt 2.2 beschrieben wurde.

Wie *Interface*-Methodenaufrufe behandelt werden sollen, lässt sich innerhalb des Frontends festlegen. Das Standardverhalten in der aktuellen Implementierung ist, dass zur Laufzeit eine Suche innerhalb der *RTTI* [1] (Siehe Kapitel 2.4.1) ausgeführt wird.

2.8. X10i

X10i ist ein X10-Compiler [5], der auf der X10-Compiler-Implementierung von IBM basiert. Dieser verwendet im Gegenteil zum Original libFirm als Compiler-Framework.

X10 ist eine objektorientierte Programmiersprache, die speziell für parallele Programmierung entwickelt wurde. So werden viele hilfreiche Konstrukte direkt von der Sprache unterstützt. Die Grundidee besteht darin, Aktivitäten auf verschiedene *Places* aufzuteilen. Die Syntax orientiert sich stark an Scala.

Weitere Informationen zur Syntax finden sich in den Sprach-Spezifikationen [17].

2.9. Bytecode2firm

Bytecode2firm ist ein Frontend für libFirm, welches Java-Bytecode in Maschinencode übersetzt. Der Vorteil von Bytecode2firm ist, dass der Code relativ einfach ist und sich Programme schnell übersetzen lassen. Dank wenigen Abhängigkeiten eignet sich Bytecode2firm optimal zum Testen und Entwickeln neuer Funktionalitäten.

[1] Runtime Type Information

3. Entwurf und Implementierung

Für diese Arbeit wurden zwei verschiedene Ansätze für Interface-Methodenaufrufe implementiert. Zum einen *Searched ITables* 2.5.2 und zum anderen *Directly Indexed ITables* 2.5.1. Diese zwei Ansätze erschienen am sinnvollsten, da sie einen guten Kosten-Nutzen Faktor haben. Weitere Ansätze wie *Bidirektionale Vertikale Methodentabellen* wurden ebenfalls diskutiert.

Aktuell verwendet liboo *Class Object Search*, welche eine Laufzeitsuche innerhalb der RTTI ausführt (Siehe Unterabschnitt 2.4.1). Dieser Aufwand ist für jeden Interface Aufruf fällig und eigentlich nicht vertretbar.

3.1. Ablauf der Implementierung

Die Implementierung erfolgte in zwei Schritten. Zuerst wurde die grundlegende Funktionalität innerhalb von Bytecode2firm (Kapitel 2.9) erstellt. Die Änderungen, die hierfür notwendig waren, wurde dann in einem zweiten Schritt nach X10i portiert.

Ziel hierbei war es, alle wesentlichen Änderungen innerhalb von liboo zu implementieren. Änderungen außerhalb sollten auf ein Minimum reduziert werden, um spätere Portierung für andere Frontends möglichst einfach und ohne größere Anpassungen zu gestalten.

Die Implementierung innerhalb von liboo teilt sich hierbei in folgende drei Teile auf:

1. Erstellung der *itables*
   Jede Implementierung die auf *itables* basiert, verwendet identische *itables*. Die Umsetzung für *Searched ITables* und *Directly Indexed ITables* ist daher komplett identisch.

2. Erstellung der *itt*'s
   Die vorher erstellten *itables* werden anschließend als Referenzen in die *itt*'s eingefügt. Für *Directly Indexed ITables* erfordern diese eine bestimmte Anordnung, für *Searched ITables* wird zusätzlich ein Identifier benötigt.

Die Referenz auf die *itt* wird in der *vtable* abgelegt. Beim späteren Zugriff auf diese wird daher eine Indirektion mehr benötigt. Alternativ könnte man diese auch direkt im Objektlayout ablegen. Hierfür müsste das Objektlayout angepasst werden, was die Kompatibilität mit anderen Frontends beeinflusst. Daher wurde dies nicht implementiert.

3. Umwandlung der `MethodSel`-Knoten in Interface-Aufrufe
   Im letzten Schritt werden die `MethodSel`-Knoten ersetzt, damit libFirm bzw. das Backend den richtigen Aufrufcode für Interface-Aufrufe generieren kann.

3.2. Rekursive Klassenhierarchien

Da wir uns innerhalb von Klassenhierarchien bewegen, haben wir es vorwiegend mit rekursiven Strukturen zu tun. Eine *Walker*-Funktion, die diese Struktur mithilfe eines Callbacks durchläuft, erwies sich als unbequem.

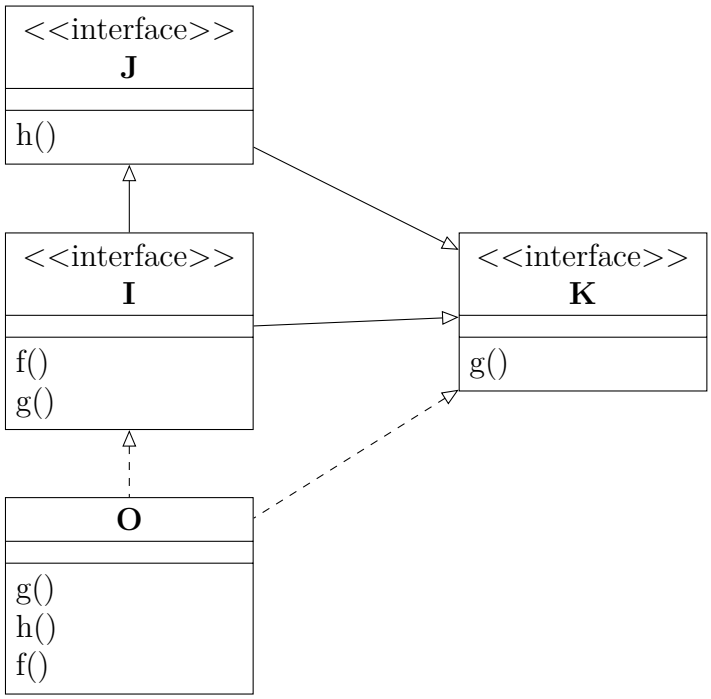


Abbildung 3.1.: Klasse, die indirekt mehrfach *K* implementiert

Daher wurde eine Funktion erstellt, welche die Menge über alle Supertypen einer Klasse bildet, über die man iterieren kann.

$$Super(p) := \{\sigma : p \leq \sigma\} \tag{3.1}$$

$$Interfaces(p) := \{i : i \in Super(p) \wedge i \textbf{ ist } Interface\} \tag{3.2}$$

$\sigma \leq \tau$ beschreibt die Halbordnung „Jedes σ-Objekt ist auch ein τ-Objekt“. Insbesondere ist $\leq$ reflexiv und daher gilt $\sigma \leq \sigma$ [9, Folie 247]. Die Menge $Super(p)$ beinhaltet sowohl Klassen als auch Interfaces. $Interfaces(p)$ ist die Einschränkung von $Super(p)$ auf Interfaces. Da es sich um Mengen handelt, sind alle Elemente auch nur einmal in diesen vorhanden.

Die rekursive Implementierungsstruktur von Interfaces erfordert, dass für alle Interfaces, die eine Klasse implementiert, *itables* erstellt werden. In unserem Beispiel (Abbildung 3.1) wären dies für die Klasse `O` *itables* für die Interfaces `I`, `J` und `K`. Für das Interface `K` wird nur eine *itable* benötigt, auch wenn diese insgesamt drei mal in der Hierarchie vorkommt.

3.3. Implementierung von *Directly Indexed ITables*

Zuerst wird die Implementierung von *Directly Indexed ITables* (Siehe Kapitel 2.5.1) betrachtet. Diese läuft wie in wie Abschnitt 3.1 beschrieben in drei Schritten ab und beginnt mit der Erstellung der *itables*.

3.3.1. Erstellung der *itable*

Die Generierung aller notwendigen Tabellen erfolgt innerhalb der `lower_oo` Funktion innerhalb von `oo.c`. In dieser werden mit Hilfe einer Walker-Funktion alle Klassen in topologischer Reihenfolge durchlaufen, d.h. eine Klasse wird erst erreicht, wenn all ihre Supertypen bereits erreicht wurden. Dies hat den Vorteil, dass der Walker bereits alle Interfaces durchlaufen hat, bevor er die zugehörigen Klassen erreicht. Die Abarbeitungsreihenfolge für Abbildung 3.1 ist somit `K` $\rightarrow$ `J` $\rightarrow$ `I` $\rightarrow$ `O`. Dadurch ist sichergestellt, dass wir jedem Interface bereits einen eindeutigen Bezeichner zugeordnet haben, bevor wir die zugehörigen Klassen erreichen. Dies ist für die Erstellung der *itt*'s wichtig (Siehe Abschnitt 3.3.2).

Für die *itables* erstellen wir für jedes Paar (K, I) eine *itable*, wobei K eine Klasse ist und $I \in Interfaces(K)$. Diese hat so viele Einträge, wie in I Methoden deklariert sind[1]. Die Reihenfolge der Methoden wird vom Interface-I, unabhängig von der zugehörigen Klasse K, vorgegeben. Zwei Paare (O, I) und (U, I) besitzen die gleiche Methodenreihenfolge in der *itable*. Die korrekte Reihenfolge der Methoden innerhalb der *itable* wird sichergestellt, indem wir ausgehend von dem Interface `I` für eine Klasse `K` alle Methoden durchlaufen und die passenden Methodendeklarationen innerhalb der Klassen $Super(K)$ suchen.

[1] Wenn I von anderen Interfaces erbt, werden diese Methoden auch berücksichtigt

Für die Klasse `O` und das Interface `I` aus Abbildung 3.1 ergibt sich beispielsweise folgende Methodenreihenfolge innerhalb der *itable*: `f()`, `g()`, `h()` (`I` → `J` → `K`)

3.3.2. Erstellung der *itt*'s

Um die *itt* für eine Klasse `K` zu erstellen, wird zuerst ermittelt, wie groß diese werden soll. Hierfür ordnet man zunächst jedem Interface eine globale eindeutig aufsteigende Nummer (*index*) zu, auf welche man über $index(I) \to \mathbb{N}_0$ zugreifen kann. Die Größe der `K`-*itt* ergibt sich, indem man den größten Index für ein Interface das K implementiert verwendet.

$$size_{itt}^{diit}(K) := max\,\{index(I) + 1 : I \in Interfaces(K)\}$$

Dadurch hat die *itt* ausreichend Felder um alle implementierten Interfaces einzutragen. Auf alle weiteren Interfaces kann aufgrund der statischen Typprüfung niemals zugegriffen werden.

Aufgrund der Abarbeitungsreihenfolge des Walkers (Von Eltern- zu Kindklassen) ist hier sichergestellt, dass alle Interfaces `I`, die eine Klasse `K` implementieren, bereits erreicht wurden, wenn die *itt* erstellt wird und $size_{itt}^{diit}(K)$ den Interfaces bereits einen Index zugewiesen hat. Die *itables* können sofort beim Erstellen in die *itt* am richtigen Index ($index(I)$) eingetragen werden. Die Adresse der *itt* wird an einem festen Index in der *vtable* abgelegt.

3.3.3. Ersetzung des MethodSel-Knoten

Zum Abschluss muss der `MethodSel`-Knoten durch passende andere Firm-Knoten ersetzt werden. Hierfür haben wir folgenden Ablauf:

1. Die *vtable* laden
2. Berechnung der Adresse der *itt* (Addition)
3. Laden der *itt*
4. Berechnung der Adresse der *itable* (Addition)
5. Laden der *itable*
6. Berechnung der Adresse der Methode (Addition)

Diese Knoten werden später vom Backend in drei `LOAD`-Befehle übersetzt. Interessant ist hier, dass nicht jeder Knoten in einen eigenen Maschinenbefehl übersetzt wird. So werden jeweils mehrere Knoten in einem einzigen `LOAD`-Befehl zusammengefasst, der sich auch um die Adressrechnung kümmert. In unserem Fall wird der `MethodSel`-Knoten in drei `LOAD`-Befehle übersetzt: `movl offset(%1), %2` (AT&T-Assembler Syntax). Je nach Architektur unterstützt der `CALL`-Befehl diese Adressberechnung ebenfalls `call *offset(%1)`.

Die Implementierung erfolgt analog zur bereits vorhandenen Implementierung der *vtable*-Aufrufe. Diese muss lediglich um eine weitere Indirektion zum Laden der *itable* ergänzt werden. Wenn der *itt*-Zeiger direkt im Objektlayout gespeichert wird, können die Schritte 1 und 2 übersprungen werden.

3.4. Implementierung von *Searched ITables*

3.4.1. Erstellung der *itable*

Die Erstellung der der *itables* erfolgt analog zur Erstellung der *itables* für den *Directly Indexed ITables*-Ansatz und kann für beide Implementierungen identisch übernommen werden. Siehe Abschnitt 3.3.1.

3.4.2. Erstellung der *itt*

Die Größe der *itt* für eine Klasse `K` ergibt sich einfach aus der Anzahl an Interfaces, die die zugehörige Klasse implementiert.

$$size_{itt}^{sit}(K) := |Interfaces(K)|$$

In welcher Reihenfolge nun die *itables* in der *itt* gespeichert werden sollen, spielt für die Funktionalität keine Rolle. Um die passende *itable* wieder zu finden wird zusätzlich zur *itable*-Referenz eine ID gespeichert, mit der man progammübergreifend Interfaces wiedererkennt. Die einfachste Art dies zu erreichen ist, den Namen des Interface als ID zu verwenden. Dies würde funktionieren, hätte allerdings den Nachteil, dass zum Vergleichen der ID's ein Stringvergleich, wie bei der *Class Object Search*, benötigt wird. Die erwartete Laufzeit wäre kürzer als bei der *Class Object Search*, vermeiden möchte man diesen Aufwand aber dennoch.

Vergibt man den Interfaces einfach aufsteigend eine Nummer, hat man das Problem, dass man diese Nummerierung über Übersetzungseinheiten hinweg nur schwer synchronisieren kann. Einfach nur aufsteigend nummerieren eignet sich hier also leider nicht. Als Alternative zu einer aufsteigenden Nummerierung kann man einen numerischen Hash aus dem Namen des Interfaces erzeugen. Dies hat aber das Problem, dass spätestens wenn Kollisionen auftreten ein Stringvergleich benötigt wird, um die Kollisionen aufzulösen.

Am besten wäre es, wenn man einen String als Identifier verwendet, den der Linker in eine passende eindeutigen Zahl umwandelt und diese über alle Übersetzungseinheiten hinweg eindeutig hält.

Ein Weg dies zu erreichen ist die Verwendung von `COMDAT`-Sections [14]. Wenn Symbole mit dem gleichen Namen in mehreren Objekt-Dateien gleichzeitig definiert sind, würde der Linker normalerweise Fehler melden. Da dies aber teilweise erwünscht oder sogar benötigt wird (z.B. für C++-Templates), kann man diese gemeinsamen Sections als `COMDAT` markieren. Das Standardverhalten ist in diesem Fall, dass alle weiteren Symbole mit dem selben Namen verworfen werden.

Dieses Verhalten können wir uns zunutze machen, indem wir ein neues globales Symbol für jedes Interface anlegen und in einer als `COMDAT` markierten Section ablegen. Diese Symbole (bzw. Adressen) verwenden wir nun als ID. Der Linker kümmert sich dann darum, dass die ID eindeutig für alle Übersetzungseinheiten ist.

Diese so erstellte ID speichern wir zusätzlich zur *itable*-Referenz in der *itt*.

3.4.3. Ersetzung des MethodSel-Knoten

Für *Searched ITables* wird der `MethodSel`-Knoten in einen `CALL` Befehl umgewandelt, der den Laufzeitservice zum Auffinden der konkreten Interface-Methode aufruft.

In unserem Fall ist dies die `oo_searched_itable_method`-Funktion. Diese erwartet eine Referenz auf das Objekt, die Interface-ID und den Index der aufzurufenden Methode innerhalb der *itable*. Das weitere Vorgehen ist dann wie in 2.5.2 beschrieben.

In Abbildung 3.2 wird das Ende der *itt* durch einen `NULL`-Eintrag am Ende der markiert. Da wir durch statische Typprüfung aber sicher sein können, dass dieser existieren muss, kann diese Bedingung auch entfallen. Dies wäre nur möglich, wenn wir einen Interface-Aufruf auf ein Objekt machen, welches dieses Interface nicht implementiert - es also keinen Eintrag in der *itt* gibt. Der hierfür nötige Cast der Referenz auf das nicht implementierte Interface, würde aber schon vor dem Interface-Aufruf fehlschlagen.

Ein weiteres Problem dieser Methode ist, dass durch eine ungünstige Sortierung der

Input : obj; id; index
Output : method
itt ← *obj.vptr.itt*
i ← 0
repeat
  if *itt[i].id* = *id* **then**
    return itt[i].itable[index]
  i ← *i* + 1
until *itt[i].id* ≠ *NULL*

Abbildung 3.2.: Searched ITable-Laufzeitservice: *obj* enthält eine Referenz auf das Objekt, *id* die ID der zu suchenden *itable* und *index* den Index der aufzurufenden Methode innerhalb der *vtable*. Da die ID eine Adresse ist, muss hier kein Stringvergleich ausgeführt werden.

itt ggf. die komplette Liste nach der passenden *itable* durchsucht werden muss. Um diesem Problem zu begegnen, wurde der Laufzeitservice testweise um eine Move2Front-Implementierung ergänzt. Diese hat bei einem Interface-Aufruf, der die gesuchte *itable* nicht in den ersten fünf[2] Einträgen gefunden hat, den Eintrag an den Anfang der *itt* verschoben. Hierfür wurde eine doppelt-verkettete Liste verwendet, die innerhalb der *itt* zusätzlich zur ID und der Referenz auf die *itable* noch zwei weitere Felder für den nächsten und vorherigen Index innerhalb der *itt* enthielt.

Da dies in Real-World Programmen aber zu keinem messbaren Geschwindigkeitsvorteil führte und der Laufzeitservice in seiner einfachen Form nicht thread-safe war, wurde dies verworfen. Die Verwendung von Mutexen oder ähnlichen Locking-Techniken würde sich des weiteren negativ auf die Laufzeit auswirken und den Vorteil, den wir durch Move-to-Front gewonnen haben, wieder kompensieren.

3.5. Bidirektionale Vertikale Methodentabelle

Die Implementierung dieses Ansatzes wurde ebenfalls in Erwägung gezogen. Bei genauerer Untersuchung dieser Methode in Zusammenhang mit liboo stellte sich aber heraus, dass die hierfür notwendigen Änderungen nicht ohne tiefere Änderungen an der kompletten *vtable*-Implementierung und weiteren Teilen des Quellcode machbar sind.

Die Hauptprobleme die sich hierbei herausstellten waren unter anderem:

- Negativ wachsende *vtable* und Verschiebung des *vptr* auf andere „Nullstelle“.

[2] Empirisch ermittelter Wert, der für viele Fälle sinnvoll erschien.

- *vtables*, die teilweise unabhängig und außerhalb von liboo erstellt wurden.

Zudem hat diese Methode den Nachteil, dass sie nicht mit getrennten Übersetzungseinheiten funktioniert und mit wachsender Interface-Anzahl auch zu sehr großen *vtables* führt.

Nach Abwägung des Aufwands und der dafür notwendigen Zeit und Umstrukturierungen, wurde entschieden, dass dieser Ansatz nicht implementiert wird. Der dafür nötige Aufwand würde den Rahmen dieser Bachelor-Thesis übersteigen.

4. Evaluation

Man betrachten nun, wie sich die Änderungen auf die Performance der mit libFirm erstellten Programme ausgewirkt hat. Hierzu teilen man die Auswertungen in zwei Bereiche ein.

Zum einen werden selbst erstellte synthetische Testfälle betrachten und zum anderen ein „Real-World“-Programm.

4.1. Anmerkung zur Ausführung/Auswertung der Testfälle

Die Zeitmessungen erfolgten auf einem Server-System mit einem Intel Core i7-2600 (3.4GHz, 8MB Cache) und 2x8GB DDR3 Arbeitsspeicher. Als Betriebssystem wurde Debian 8.2 mit Kernel 3.16.0-4-amd64 eingesetzt. Die getestete Bytecode2firm-Version basiert auf 985467c[1], libfirm b068edd[2] und liboo e8c7eae[3]. Für X10i 705ec8f8f5f[4] wurde libfirm 8fa3e36[5] und liboo e8c7eae[6] verwendet.

Die synthetischen Testfälle wurden jeweils so angelegt, dass das Referenzprogramm ohne verbesserte Interface-Aufrufe eine Ausführungszeit von etwa 10sec hat. Dadurch kann man sicherstellen, dass der wesentliche Teil der Programmausführung für Interface-Aufrufe verwendet wird.

Für die Zeitmessung wurde ein selbstgeschriebenes Java-Programm verwendet, das automatisiert unsere Testprogramme mit den verschiedenen Interface-Aufrufarten kompiliert und mehrfach ausführt. Für kurze schnelle Tests bereits kompilierter Programme wurde folgender Bash-Befehl verwendet:

[1] 985467c83da209a566f93f0ad05e819e0445500a
[2] b068edd11a5376339d6d57e0528a33e3fe409738
[3] e8c7eaed357deb5d813e546c04448908fd71b33a
[4] 705ec8f8f5fbe49050c6be1e6875629bae7a1fee
[5] 8fa3e3649fd61e23c4eb260273ae4a9689fe19b0
[6] e8c7eaed357deb5d813e546c04448908fd71b33a

```
for run in {1..10}; do time ./Program; done
```

Die Testfälle wurden pro Testlauf 10-mal ausgeführt. Dabei wurde jeweils das beste Testergebnis verwendet, da dieses der theoretisch minimal erreichbaren Ausführungszeit am nächsten kommt. Alternativ könnten man auch den Mittelwert über alle Ausführungszeiten verwenden. Dies hätte aber den Nachteil, das Ausreißer nach oben, aufgrund von System-Hintergrundaktivitäten, ebenfalls in die Auswertung einfließen würden.

Die Reduktion berechnen man mit folgender Formel (Ergebnis in %)

$$Reduktion = \frac{Referenz - Messung}{Referenz} * 100$$

Für den Speed-Up wird folgende Formel verwendet

$$Speedup = \frac{Messung}{Referenz}$$

Hierbei ist jede $Reduktion > 0$, bzw $Speedup > 1$ eine Verbesserung.

Gemessen wurde bei den synthetischen Testfällen teilweise auch zusätzlich der Move-to-Front-Ansatz (M2F) für *Searched ITables*, da dieser das Problem der temporären Lokalität umgeht.

4.2. Synthetische Testfälle

Zunächst wird den Extremfall für *Class Object Search* betrachtet. Es wird eine Klasse mit sehr vielen Methoden, welche von sehr vielen anderen Klassen (mit vielen Methoden) erbt und die Methodennamen jeweils sehr lange sind konstruiert.

Für die Tests wurde das Programm aus Listing A.1 mit $N = 10$, $M = 10$, $K = 10$ und $X = 10^7$ ausgeführt. Siehe Tabelle 4.1.

Bereits bei diesen noch relativ kleinen Werten für M, N und X zeigt sich die Schwäche von *Class Object Search*. Erhöht man die Anzahl der Methoden oder Vererbungen auf 1000 oder noch mehr, würde der Effekt noch viel stärker hervortreten. Die Laufzeit von *Directly Indexed ITables* hingegen dürfte sich durch diese Änderungen kaum von den aktuellen Messwerten unterscheiden.

Die Frage ist nun, wie sich ein Test verhält, der so geschrieben wurde, dass *Class Object Search* sehr gut abschneiden sollte. Dieser Test besteht aus einer Klasse, die ein Interface

Methode	*min*(*Laufzeit*)	Reduktion	Speed-Up
Class Object Search	$7,31s$		
Searched ITables	$0,08s$	$98,88\%$	89.11
Searched ITables (M2F)	$0,1s$	$98,67\%$	75.33
Directly Indexed ITables	$0,04s$	$99,51\%$	202.97

Tabelle 4.1.: Testlauf zu Listing A.1 für $M = 10$, $N = 10$, $K = 10$, $X = 10^7$.

implementiert, welches eine Methode beinhaltet. (Siehe Listing A.3). *Class Object Search* findet die aufzurufende Methode hier sehr schnell. Man sollte also davon ausgehen, dass die Messwerte (Siehe Tabelle 4.2) hier sehr nahe beieinander liegen.

Methode	*min*(*Laufzeit*)	Reduktion	Speed-Up
Class Object Search	$3,334s$		
Searched ITables	$0,525s$	$84,3\%$	6.37
Searched ITables (M2F)	$0,647s$	$80,65\%$	5.17
Directly Indexed ITables	$0,188s$	$94,38\%$	17.79
vtable-Aufruf	$0,188s$	$94,38\%$	17.79

Tabelle 4.2.: Testlauf zu Listing A.3.

Das Ergebnis entsprach aber nicht diesen Erwartungen. *Class Object Search* findet bereits in der ersten betrachteten Klasse, die gesuchte Methode. Dies entspricht dem selben Aufwand zu Interfaces, welcher auch *Searched ITables* hat. Dass das Ergebnis trotzdem deutlich schlechter ausfällt, liegt also an der Suche der Methode innerhalb der Klasse. Anders als erwartet ist `foo` nicht die erste Methode in der Liste, da `Test` implizit von `Object` erbt und daher auch dessen Methoden enthält.

Selbst bei einfachsten Testfällen ist die neue Implementierung also um den Faktor 6 (*Searched ITables*) oder sogar 17 (*Directly Indexed ITables*) schneller als die naive Suche.

Zum Vergleich wurde dieser Test auch mit einem virtuellen Methodenaufruf ausgeführt (Siehe Tabelle 4.2). Es zeigt sich, dass der Geschwindigkeitsunterschied zwischen *Directly Indexed ITables* und virtuellen Methodenaufrufen hier vernachlässigbar klein ist. In der gegebenen Größenordnung ließ sich kein Unterschied messen. Dies liegt wahrscheinlich daran, dass sich die komplette *itt* und *itable* im Prozessor-Cache befindet und daher die Aufrufzeit vor allem von dem `CALL`-Befehl dominiert wird.

Interessanterweise ist der Move-to-Front-Ansatz hier sogar spürbar langsamer als der native *Searched ITables*-Ansatz. Dies lässt sich durch die zusätzliche Logik erklären.

Es bleibt noch die Frage, wie sich *Searched ITables* im Worst-Case verhält. Hierzu konstruiert man eine Klasse, die sehr viele Interfaces Implementiert. Der Interface-Methodenaufruf wird nun auf dem letzten Interface in der Liste ausgeführt. Der Code dazu befindet sich in Listing A.2.

Damit der Test mit Tabelle 4.2 vergleichbar ist, wurde er mit $N = 10$ und $N = 100$ ausgeführt. Siehe Tabelle 4.3.

Methode: $N = 10$	$min(Laufzeit)$	Reduktion	Speed-Up
Class Object Search	$3,334s$		
Searched ITables	$2,415s$	$27,56\%$	1.38
Directly Indexed ITables	$0,188s$	$94,38\%$	17.73
Methode: $N = 100$			
Class Object Search	$3,334s$		
Searched ITables	$22,125s$	$-563,62\%$	0.15
Directly Indexed ITables	$0,188s$	$94,38\%$	17.73

Tabelle 4.3.: Testlauf zu Listing A.2 für $N = 10$ und $N = 100$

Wie erwartet zeigt sich, dass sich *Class Object Search* und *Directly Indexed ITables* nicht von der Anzahl an Interfaces beeinflussen lassen. Ihre Laufzeiten blieben für beide Messungen identisch. *Class Object Search* hängt nur von den Methoden und Klassen in der Hierarchie ab; *Directly Indexed ITables* hat gar keine Abhängigkeiten und genauso wie *vtable*-Aufrufe konstante Laufzeit.

Im Gegenteil dazu stieg die Laufzeit von *Searched ITables* proportional zur Anzahl an implementierten Interfaces. Dies entspricht auch dem erwarteten Verhalten.

Diese Messung wurde auch mit aktiviertem Move-to-Front ausgeführt. Die Ergebnisse entsprachen dabei den gleichen wie bei Tabelle 4.2.

4.3. Real-World Test

Zuletzt wird noch ein „Real-World“-Programm betrachtet, um den tatsächlichen nutzen in einem realen Programm auszuwerten. Hierfür wird das X10i-Programm Multigrid [4] verwendet. Dieses berechnet, wie sich ein Hitzepunkt auf einer Metallplatte ausbreitet.

Innerhalb von dem Testprogramm wurde kein besonderer Programmierstil verwendet (z.B. besonders viel/wenige Interface-Aufrufe) und es wurde sich an die X10 Program-

mierparadigmen gehalten. Um sinnvolle Messergebnisse zu erhalten, wurde der Timesteps-Parameter von Multigrid angepasst. Hierbei wurden Messreihen für die Timesteps $N \in 200, 400, 600$ angelegt. Siehe Tabelle 4.4.

Methode: $N = 200$	$min(Laufzeit)$	Reduktion
Class Object Search	$4,403s$	
Searched ITables	$4,321s$	$2,54\%$
Searched ITables (M2F)	$4,312s$	$2,75\%$
Directly Indexed ITables	$4,305s$	$2,91\%$
Methode: $N = 400$		
Class Object Search	$8,857s$	
Searched ITables	$8,651s$	$2,33\%$
Directly Indexed ITables	$8,642s$	$2,43\%$
Methode: $N = 600$		
Class Object Search	$13,160s$	
Searched ITables	$12,983s$	$1,34\%$
Directly Indexed ITables	$12,837s$	$2,45\%$

Tabelle 4.4.: Testlauf für Multigrid mit $N = 600$

Es zeigt sich, dass sowohl *Searched ITables* als auch *Directly Indexed ITables* $> 2\%$ schneller sind, als der naive Ansatz mit *Class Object Search*. Angesichts der Tatsache, dass man für diese Geschwindigkeitsverbesserung keine Änderungen an dem Quelltext von Multigrid vornehmen mussten, ist diese Verbesserung beachtlich.

Einige Sprachkonstrukte von X10 werden in Interfaces transformiert. Ein Funktionstyp wird beispielsweise in ein Interface mit einer einzigen `apply`-Funktion umgewandelt. Die für parallele Ausführung wichtigen `async`- und `at`-Blöcke werden in Funktionsobjekte transformiert. Deren Verwendung ist also jedes mal mit einem Interface-Aufruf verbunden. Dies erklärt, warum selbst X10-Programme, die fast keine Interface-Aufrufe verwenden, von den verbesserten Interface-Aufrufen profitieren.

4.4. Getrennte Übersetzungseinheiten

Um die Funktionalität von getrennten Übersetzungseinheiten zu testen, musste mehr Aufwand betrieben werden. Zum Zeitpunkt dieser Arbeit unterstützen weder Bytecode2Firm noch X10i getrennte Übersetzungseinheiten.

Es wurden zwei Klassen `A` und `B` erstellt, die jeweils Interface `I` mit einer Funktion `foo()` implementieren. Klasse `A` enthält zusätzlich Code, der die Interface Funktion `foo()` sowohl für `A` als auch `B` aufruft.

Es wurden nun beide Klassen getrennt von Bytecode2Firm übersetzt und die dabei entstehenden Assembler-Dateien von Hand nachbearbeitet. In *A.s* wurde alles von `B` entfernt und in *B.s* wurde nur der Code von `B` beibehalten. Die so entstandenen Dateien wurden anschließend vom Linker zusammengefügt, wobei es keine Probleme gab. Das Programm zeigte beim Ausführen das gewünschte Verhalten.

5. Fazit und Ausblick

5.1. Zusammenfassung

In dieser Arbeit wurde eine funktionsfähige Implementierung für *Searched ITables* und *Directly Indexed ITables* für Firm erstellt, um Interface-Funktionsaufrufe zu beschleunigen.

Es wurde erklärt, warum sich die naive Implementierung so schlecht verhält, welche Alternativen es zu dieser gibt und warum sich diese viel besser verhalten.

Es stellte sich heraus, dass *Directly Indexed ITables* ein sehr schneller und effizienter Ansatz ist, der in etwa das gleiche Laufzeitverhalten aufweist, wie Methoden, die mit dem *vtable*-Mechanismus aufgerufen werden. Allerdings lässt sich dieser Ansatz nur verwenden, wenn man auf getrennte Übersetzungseinheiten verzichten kann.

Als Alternative zu *Directly Indexed ITables* wurde *Searched ITables* vorgestellt. Dieser Ansatz bewegt sich im Laufzeitverhalten in der gleichen Größenordnung wie *Directly Indexed ITables*, kann aber in Extremfällen auch deutlich langsamer sein. Allerdings funktioniert dieser Ansatz auch mit getrennten Übersetzungseinheiten. Er ist also ein guter Trade-off, wenn man sowohl schnelle Interface-Aufrufe, als auch getrennte Übersetzungseinheiten benötigt.

5.2. Zukünftige Arbeiten

Ein nächster Schritt um den *Searched ITables*-Ansatz noch weiter zu optimieren wäre eine Verbesserung der Sortierung der *itt*. Hierfür bestehen verschiedene Möglichkeiten. Zum einen kann durch statische Codeanalyse ermittelt werden, welche Interfaces oft verwendet werden. Man kann zum Beispiel die Anzahl an Methodenaufrufen zählen, oder wie oft ein Interface-Methodenaufruf in einer potentiell großen Schleife aufgerufen wird. Interfaces, die oft benötigt werden, werden in der *itt* weiter oben angeordnet, andere weiter unten. Dadurch wird sichergestellt, dass nicht die komplette Liste durchsucht werden muss, um einen oft verwendeten Interface-Aufruf zu finden. Alternativ kann man

durch mehrfaches Kompilieren mit Profilinginformationen die heißen Interfaces ermitteln und diese in der *itt* am Anfang anordnen. Dies verspricht genauere Ergebnisse als eine statische Codeanalyse.

Ebenfalls besteht die Möglichkeit, den Move-to-Front-Ansatz erneut mit einer Thread-Safe-Methode zu implementieren. Durch eine geschickt angelegte Liste oder andere Techniken, kann man ggf. auch die Verwendung von Locking-Mechanismen verzichten, was diesen Ansatz wieder attraktiv machen würde. Eine Möglichkeit hierfür ist, die *itt* am Anfang um einen leeren Bereich zu erweitern. Diesen füllt man mit den Indizes oft benötigter Interfaces. Das Schreiben in dieser Tabelle erfolgt mit einer atomaren CAS-Operation (Compare & Swap) [7] Inkonsistenzen durch Race-Conditions lassen sich so vermeiden. Beim suchen nach einer *itable* werden die so gespeicherten Indizies zuerst geprüft und falls kein Treffer gefunden wird, in der restlichen *itt* gesucht. Einträge werden hierdurch ggf. mehrfach betrachtet.

Weiterhin gibt es noch andere Techniken für Interface-Methodenaufrufe, die in dieser Arbeit nicht behandelt wurden. So kann beispielsweise noch geklärt werden, ob und wie sich IMT's (Interface Method Table) [1] in Firm verwenden lassen. Diese Methoden verwendet eine Hash-Table um die richtigen Methoden wiederzufinden. Hierzu wird aus Methodennamen und Signatur ein numerischer Hash erzeugt, den man als Offset für die IMT verwendet. Für Kollisionen wird in der IMT eine *Selector*-Funktion referenziert, welche die passende Methode aufruft.

Aufgrund der Messergebnisse aus Kapitel 4 hat sich herausgestellt, dass *Bidirektionale Vertikale Methodentabellen* keinen nennenswerten Mehrwert gegenüber *Directly Indexed ITables* haben. Die Geschwindigkeitsunterschied von *vtable*-Methodenaufrufen und *Directly Indexed ITables*-Methodenaufrufen ist so minimal, dass sich eine Implementierung kaum lohnt. *Directly Indexed ITables* bieten darüber hinaus den Vorteil, dass die *vtable* nicht größer wird.

Literaturverzeichnis

[1] B. Alpern, A. Cocchi, S. Fink, and D. 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, New York, NY, USA, 2001. ACM.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Progranzming Language*. PhD thesis, Department of Computer Science, University of Copenhagen, May 1994.

[3] W. M. Bacon, D. F. and K. Zadeck. Rapid type analysis for c++. Technical report, IBM Thomas J. Watson Research Center, 1996.

[4] H.-J. Bungartz, C. Riesinger, M. Schreiber, G. Snelting, and A. 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.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, Oct. 2005.

[6] 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, New York, NY, USA, 1995. ACM.

[7] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[8] D. Novillo. Tree ssa a new optimization infrastructure for gcc. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193. Citeseer, 2003.

[9] Pr. Dr.-Ing. Gregor Snelting. Fortgeschrittene objektorientierung, 2015.

[10] 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.

[11] Spring Framework 4.2.2.RELEASE API. Spring Framework 4.2.2.RELEASE API. http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/context/support/GroovyWebApplicationContext.html, retrieved on 2015-11-13.

[12] B. 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.

[13] B. Stroustrup. Multiple inheritance for c++. *Computing Systems, Vol. 2 - No. 4*, Year.

[14] This document was developed jointly by an informal industry coalition consisting of (in alphabetical order) CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Additional contributions were provided by a variety of individuals. Itanium C++ ABI. https://mentorembedded.github.io/cxx-abi/abi.html, retrieved on 2015-11-20.

[15] M. Trapp. *Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.* PhD thesis, University of Karlsruhe, Faculty of Informatik, Oct. 2001.

[16] M. A. und Christian von Roques. Entwurf und realisierung eines sather-k Übersetzers. Master's thesis, University of Karlsruhe, Faculty of Informatik, dec 1996.

[17] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 Language Specification. http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf, retrieved on 2015-11-18.

[18] C. W. A. Wawersich. *KESO: Konstruktiver Speicherschutz für Eingebettete Systeme.* PhD thesis, Technische Fakultät der Universität Erlangen-Nürnberg, 2009.

Erklärung

Hiermit erkläre ich, Philipp Serrer, dass ich die vorliegende Bachelorarbeit 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. Code

Listing A.1: Testprogramm um den denkbar schlechtesten Fall für *Class Object Search* zu testen. Dabei sind M, N, K und X frei wählbar. Je höher M, N oder K gewählt sind, umso schlechter ist das Laufzeitverhalten der *Class Object Search*

```
1  interface I { void
2      SehrLangerMethodenName....M(int Parameter1, ..., int
          ParameterK);
3  }
4
5  class KlasseN {
6      void SehrLangerMethodenName....1(int Parameter1, ...,
           int ParameterK) { }
7      ...
8      void SehrLangerMethodenName....M(int Parameter1, ...,
           int ParameterK) { }
9  }
10 class Klasse[N-1] extends KlasseN {
11     void SehrLangerMethodenName....1(int Parameter1, ...,
           int ParameterK) { }
12     ...
13     void SehrLangerMethodenName....[M-1](int Parameter1,
          ..., int ParameterK) { }
14 }
15 ...
16 class Klasse1 extend Klasse2 implements I {
17     void SehrLangerMethodenName....1(int Parameter1, ...,
           int ParameterK) { }
18     ...
19     void SehrLangerMethodenName....[M-1](int Parameter1,
          ..., int ParameterK) { }
20 }
21
```

```
22 public class Test() {
23     public static void main(String[] args) {
24         I test = new Klasse1();
25         for (int i = 0; i < X; i++) {
26             test.SehrLangerMethodenName...M(...);
27         }
28     }
29 }
```

Listing A.2: Testprogramm um den denkbar schlechtesten Fall für *Searched ITables* zu testen

```
1  interface Interface1 { ... }
2  ...
3  interface InterfaceN { foo(); }
4
5  public class Test implements Interface1, ..., InterfaceN
       {
6      ...
7      public void foo() { }
8
9      public static void main(String[] args) {
10         InterfaceN test = new Test();
11         for (int i = 0; i < X; i++) {
12                 test.foo();
13         }
14     }
15 }
```

Listing A.3: Testprogramm bei dem sich die Class Object Search sehr gut verhalten sollte.

```
1 interface I { void foo(); }
2 public class Test implements I{
3     public void foo() { }
4     public static void main(String[] args) {
5         I test = new Test();
6         for (int i = 0; i < 10000000; i++) test.foo();
7     }
8 }
```