Fachbereich 12 – Informatik und Mathematik
Software Engineering und Programmiersprachen

Masterarbeit

Optimierung durch Auswahl von String-Repräsentationen im Java-Bytecode

Markus Wondrak

11. November 2014

Eidesstattliche Erklärung

Ich versichere an Eides statt durch meine eigene Unterschrift, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen genommen sind, als solche kenntlich gemacht haben. Die Versicherung bezieht sich auch auf in der Arbeit gelieferte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

(Markus Wondrak)

Danksagung

Ich danke meinem Schatz, dass er meine Schreiberei so lange ertragen hat und mich wo immer möglich unterstützte.

Weiterhin möchte ich meinem Betreuer danken, der immer ein offenes Ohr für mich hatte.

Zusammenfassung

Diese Masterarbeit untersucht, ob es mittels möglich ist statischer Code Analyse und automatisierter Transformationen von Programmen eine Optimierung des Java-`String` Typs zu realisieren. Dabei wurde ein System entwickelt, das es ermöglicht anhand von definierten Regeln verwendete Datentypen in einem Programm durch optimierte Alternativen zu ersetzen. Die Auswertung am Ende der Arbeit zeigt, dass eine Optimierung mit den gewählten Datentypen und Einschränkungen des Algorithmus nicht möglich ist. Es werden aber Weiterentwicklungen skizziert, die die aufgetretenen Probleme angehen.

Inhaltsverzeichnis

1 Einleitung **1**

2 Werkzeuge **3**
2.1 Java Bytecode 3
2.2 WALA 4
2.2.1 IR 5
2.2.2 Shrike 6

3 Optimierte Stringtypen **7**
3.1 SubstringString 7
3.2 StringListBuilder 8

4 Analyse **10**
4.1 Datenstrukturen 10
4.1.1 Datenfluss Graph 10
4.1.2 Label 13
4.1.3 Konventionen 18
4.2 Algorithmus 18
4.2.1 Motivation 18
4.2.2 Die Bubble 19
4.2.3 Umsetzung des Algorithmus 20
4.2.4 Umgang mit mehreren Labels 22
4.2.5 Phi-Knoten 23

5 Transformation **25**
5.1 Lokale Variablen 25
5.1.1 Optimierte Variablen 25
5.1.2 Variablen zu Value Numbers 25
5.2 Bytecode Generierung 28
5.2.1 Informationen des TypeLabels 28
5.2.2 Konvertierung 29

5.2.3 Optimierung 31
5.3 Schwierigkeiten 32
5.3.1 Innere Archive (JARs) 32
5.3.2 Typisierung von generiertem Bytecode 32

6 Auswertung **34**
6.1 Vorraussetzungen 34
6.1.1 Java Microbenchmarking Harness 34
6.1.2 Test Durchführungen 35
6.2 Benchmarks 36
6.2.1 Example Parser 36
6.2.2 Xalan 38

7 Fazit **50**

Abbildungsverzeichnis

2.1 Java Bytecode Beispiel 4

4.1 Beispiel einer *type* Datei, anhand der SubstringString Definition 17

5.1 Lokale Variable für Definition 26

6.1 Example Parser 36
6.2 Ergebnis des Example Parser Benchmark 37
6.3 Ergebnis des checkAttribQName Benchmark 39
6.4 Ergebnis des instantiateURI Benchmarks 40
6.5 Ergebnis des xNumberToString Benchmarks (positive Dezimalzahl) 43
6.6 Ergebnis des xNumberToString Benchmarks (negative Dezimalzahl) 44
6.7 Ergebnis des xNumberToString Benchmarks (positive Ganzzahl) 45
6.8 Ergebnis des xNumberToString Benchmarks (Zahl mit negativem Exponent) 46
6.9 Ergebnis des XPath Compile Benchmarks 48

Algorithmenverzeichnis

1 Erstellung des Datenflussgraphen 11
2 Vererbung des Labels 21
3 labelConnectedRefs 22

4 Simulation des Stacks 27
5 needsConversationTo(Label) 30

1 Einleitung

Die Verwendung von optimierten Datentypen beim Entwickeln eines Programms ist schwierig. Die eingesetzten Implementierungen von Standarddatentypen decken zumeist allgemeine Fälle ab, sind aber nicht optimal für spezielle Szenarien. Zudem sind spezielle optimierte Datentypen den Anwendern meist nicht oder zumindest nur begrenzt bekannt. So wird bei der Entwicklung von Software einer aus den Datentypen gewählt, die dem Entwickler bekannt sind. Dabei ist nicht nur die fehlende Kenntnis über diese optimierten Typen ein Grund für das Übergehen eben dieser. Die Entscheidungsfindung bis hin zur Wahl für oder gegen einen speziellen Typ verzögert die Arbeit während der Entwicklung von Systemen.

Ein weiteres Problem ergibt sich bei der Betrachtung von Altsystemen. Bei bereits lang bestehenden Anwendung, besteht das Problem der Wartbarkeit von eingesetzten Datentypen. Durch die Weiterentwicklung von eingesetzten Bibliotheken können Optimierungspotentiale erst nach der aktiven Entwicklung von einzelnen Modulen entstehen. Hierbei ergibt sich wieder das Problem, dass diese Potentiale bekannt und auch richtig umgesetzt werden müssten.

Ziel dieser Arbeit ist es daher zu untersuchen, ob sich Programme durch Auswahl und Substitution von alternativen Datentypen automatisiert optimieren lassen. Zu diesem Zweck soll ein System erstellt werden, das durch statische Code Analyse und automatische Transformation eine Optimierung eines Programms durchführt [1].

Diese Untersuchung kann allerdings keine allumfassende Auswertung bereitstellen und wird sich auf die Optimierung der Laufzeit des `java.lang.String` Datentyps der *Java* Plattform und der Programmiersprache *Java* beschränken. Nach der Auswertung soll eine Aussage darüber möglich sein, ob diese Optimierungen möglich sind. Daher sollen die beiden Hypothesen untersucht werden:

[1] Der Quellcode zu dem System (inklusive Tests) sowie die Benchmarks und Auswertungsskripte sind unter `http://github.com/wondee/faststring` zu finden.

1. Der String Datentyp bietet viele Möglichkeiten der Optimierung.
2. Das automatische Ersetzen von Standard-Datentypen durch alternative Datentypen führt zu einem Performance Gewinn.

Im Rahmen dieser Arbeit soll ein System erstellt werden, das String Operationen in einem gegebenen Java Programm mit einer entsprechenden optimierten Version auf Basis des Java Bytecodes ersetzt. Dabei soll das System anhand statischer Code Analyse jene Stellen lokalisieren, an denen eine Optimierung angewendet werden kann, und mittels der Transformation des Bytecodes des Programms diese Optimierungen anwenden. Als Ergebnis wird ein lauffähiges Programm erwartet, dass bei gleicher Eingabe eine geringere Ausführungszeit besitzt als das originale Programm.

Es existieren Arbeiten, die Optimierungen ebenfalls durch die Auswahl von alternativen Datentypen anstreben [1, 2, 3]. Diese setzen entgegen der in dieser Arbeit verwendeten statischen Code Analyse auf eine dynamische und beschränken sich auf die Auswahl von Container Typen, die zur Laufzeit ersetzt werden, um eine Optimierung des Programms zu erreichen.

In Kapitel 2 werden die Werkzeuge und Grundlagen beschrieben, auf denen das in dieser Arbeit entwickelte System aufbaut. Darauf folgt Kapitel 3, welches die optimierten String Typen vorstellt die in dieser Arbeit entwickelt wurden. Kapitel 4 erläutert den Analyse Prozess des Systems, indem die Datenstrukturen sowie der Algorithmus für die statische Code Analyse beschrieben wird. In Kapitel 5 wird basierend auf den Analyse Ergebnissen die Transformation des Bytecodes präsentiert. Kapitel 6 widmet sich der Auswertung der Tests in Form von Benchmarks und begründet die Ergebnisse. Kapitel 7 zieht ein Fazit und beschreibt mögliche Weiterentwicklungen des Systems.

2 Werkzeuge

In den folgenden Abschnitten sollen die verwendeten Werkzeuge kurz vorgestellt werden. Dabei handelt es sich sowohl um den Java Bytecode, als auch um die Software Bibliothek *WALA*, auf deren API das in dieser Arbeit entwickelte System basiert.

2.1 Java Bytecode

Die Plattformunabhängigkeit, die in Java geschriebenen Programmen zugesprochen wird, ist vor allem mit der Rolle der Java Virtual Machine zu erklären. Java Programme werden in einen Zwischencode, den Java Bytecode, übersetzt, welcher von der System spezifischen JVM ausgeführt wird. Dabei ist Programmiersprache Java nicht die einzige in Bytecode übersetzbare Sprache. Es existieren neben den bekanntesten Scala, Jython, Groovy, JavaScript noch viele weitere. Einmal in Bytecode übersetzt können in diesen Sprachen geschriebene Programme auf jeder der Java Spezifikation entsprechenden JVM ausgeführt werden.

Bytecode ist eine Sammlung von Instruktionen, welche durch *opcodes* von 2 Byte Länge definiert werden. Zusätzlich können noch 1 bis n Parameter verwendet werden. Die Sprache ist Stack-orientiert was bedeutet, dass von Operationen verwendete Parameter über einen internen Stack übergeben werden. Als Beispiel dient der folgende Bytecode:

```
ICONST 5     // legt den konstanten int Wert 5 auf den
   Stack
ILOAD 1      // läd die lokale integer Variable 1 und
   legt sie auf den Stack
IADD         // addiert die ersten beiden Werte auf dem
   Stack und legt das Ergebnis auf den Stack
ISTORE 2     // speichert den Wert auf dem Stack in der
   Variable 2
```

Abbildung 2.1: Java Bytecode Beispiel

Dabei existiert der Stack nur als Abstraktion für den eigentlichen Prozessor im Zielsystem. Wie die jeweilige JVM den Stack in der Ziel Plattform umsetzt ist nicht definiert. Die Instruktionen lassen sich in folgende Kategorien einordnen:

- Laden und Speichern von lokalen Variablen (`ILOAD`, `ISTORE`)
- Arithmetische und logische ausdrücke (`IADD`)
- Object Erzeugung bzw. Manipulation (`NEW`, `PUTFIELD`)
- Stack Verwaltung (`POP`, `PUSH`)
- Kontrollstruktur (`IFEQ`, `GOTO`)
- Methoden Aufrufe (`INVOKEVIRTUAL`, `INVOKESTATIC`)

2.2 WALA

Bei *WALA* [4] handelt es sich um die "T.J. Watson Library for Analysis", eine ehemals von IBM entwickelte Bibliothek für die statische Code Analyse von Java- und JavaScript Programmen. Das Framework übernimmt dabei das Einlesen von *class* Dateien und stellt eine Repräsentation, die sogenannte *Intermediate Representation*, des Bytecodes zur Verfügung. Diese IR stellt die zentrale Datenstruktur dar und soll in diesem Abschnitt detailliert beschrieben werden.

Für die Manipulation des Bytecodes existiert innerhalb des Frameworks ein Unterprojekt, das diese Aufgabe übernimmt: Shrike. Im Zweiten Abschnitt soll diese API kurz vorgestellt werden.

2.2.1 IR

Die *Intermediate Representation* (IR) ist eine Abstraktion zum Stack basierten Bytecode. Ein IR ist in *Single Static Assignment Form*, welche sich dadurch auszeichnet, dass jeder Variablen **einmal** ein Wert zugewiesen wird. Zusätzlich besteht die IR aus dem Kontrollflussgraphen der Methode, welcher wiederum aus *Basic Blocks* (den Knoten) zusammengesetzt ist. Ein Basic Block ist eine Zusammenfassung von aufeinander folgenden Instruktionen, welche in jedem Fall nacheinander ausgeführt werden.

Die Variablen innerhalb des IRs nennt WALA *value numbers*. Diese beziehen sich auf eine Referenz, allerdings kann sich eine Referenz auf mehrere tatsächliche *value number*s beziehen. Dies folgt aus der SSA Form, wird einer Variable im Bytecode mehr als einmal ein Wert zugewiesen, werden diese mehrfachen Zuweisungen in der SSA Form durch das Einführen einer neuen *value number* entfernt. Die Operationen innerhalb des IRs verwenden ausschließlich *value number*s um Instruktionen des Bytecodes darzustellen.

Da die Zwischendarstellung eine Abstraktion zum Stack darstellen soll, werden alle Operationen, die den Stack betreffen(wie z.B. `LOAD`, `STORE`, `PUSH` oder `POP`) nicht in dieser Repräsentation dargestellt. Dabei werden die Bytecode Indices der übrig gebliebenen Instruktionen berücksichtigt und alle anderen Stellen in dem beinhaltenden Array mit `null` Werten aufgefüllt. Instruktionen werden von Objekten vom Typ `SSAInstruction` und dessen Untertypen dargestellt.

Die Verwaltung der *value number*s wird von einem Typ namens `SymbolTable` übernommen. Es kommt bei der IR Erstellung zum Einsatz, wenn bei der Simulation des Bytecodes neue Variablen verwendet werden, um neue *value number*s zu erzeugen.

Aufgrund der SSA Form der IR lässt sich für jede *value number* genau eine *Definition* bestimmen. Zu diesem Zweck bietet WALA den Typ `DefUse` an, welcher für jedes IR-Objekt erstellt werden kann. Er ermöglicht einen einfachen Zugriff auf die Instruktion, die eine *value number* erzeugt (*Definition* einer *value number*; z.B. als Rückgabe aus einem Methodenaufruf), und auf die Menge an Instruktionen, die diese *value number* verwenden (*Use* einer *value number*; z.B. Parameter in einem Methodenaufruf oder als Rückgabewert in einem `RETURN` Statement).

Besitzt ein Block im Kontrollflussgraphen mehrere eingehende Kanten und werden aus diesen Vorgängern Referenzen in den Geltungsbereich des Blockes übernommen, die in diesem Block über den selben Zeiger referenziert werden,

so werden in SSA Form sogenannten ϕ Funktionen verwendet. Eine Instruktion der Form $v_3 = \phi(v_1, v_2)$ sagt aus, dass im folgenden Programmfluss die Referenz v_3 sowohl v_1, als auch v_2 sein kann. Da die statische Code Analyse nicht feststellen kann, von welchem Block aus dieser Block betreten wurde, werden ϕ-Funktionen verwendet, um die Zusammenführungen von mehreren Variablen aus Vorgängerblöcken darzustellen.

Das IR und das dazugehörige DefUse Objekt werden in dem systeminternen Datentyp `AnalyzedMethod` zusammengefasst.

Anpassungen

In WALA werden beim Erstellen des IR für alle Konstanten mit demselben Wert dieselbe *value number*s erzeugt. Da für die *Analyse* verschiedenen Referenzen getrennt untersucht werden, wird für die Erzeugung einer *value number* für eine Konstante der eingebaute caching Mechanismus umgangen.

Darüber hinaus ist für die *Transformation* die Information nötig, an welcher Stelle im Bytecode eine entsprechende Konstante erzeugt wird (z.B. mittels `LCD`). Um dies zu erreichen, wurde der Bytecode Index während der Durchlaufens der Instruktionen innerhalb der `SymbolTable` gespeichert, sodass der Index beim Klienten des IRs zur Verfügung steht.

Da diese Änderungen nicht in den Haupt-Branch von WALA eingepflegt werden durften, benötigt das System den Fork des WALA Projektes [1].

2.2.2 Shrike

Shrike ist ein Unterprojekt innerhalb des WALA Frameworks. Shrike übernimmt das Lesen und Schreiben von Bytecode aus bzw. in *class* Dateien. Dabei wird es zum einen beim Erstellen eines IR aus einer Methode verwendet, zum Anderen bietet es eine auf Patches basierende API an, um den Bytecode einer eingelesenen Methode zu verändern. Dies geschieht über das Einfügen von `Patch`es, welche über einen entsprechenden `MethodEditor` überall im Bytecode einer Methode eingefügt werden, oder auch ursprüngliche Instruktionen komplett ersetzen. Zusätzlich enthält es einen `Verifier`, der erzeugten Bytecode überprüft, so dass ungültige Stack Zustände oder Typfehler noch während der Manipulation erkannt werden können.

In dem von mir entwickelten System werden alle Bytecode Manipulationen mit Hilfe von Shrike umgesetzt.

[1] Dieser ist unter `http://github.com/wondee/WALA` zu finden.

3 Optimierte Stringtypen

Im Rahmen dieser Arbeit wurden zwei alternative String Repräsentationen erstellt, die Optimierungen der ursprünglichen `java.lang.String` API darstellen. Diese Typen umgehen das Design der String Repräsentationen in Java, die *nicht veränderbare* Objekte darstellen. Durch diesen Umstand führen alle Manipulationsoperationen auf String Typen dazu, dass die Daten, auf denen diese Objekte basieren (ein `char` Array, dass die einzelnen Zeichen der Zeichenkette hält) kopiert werden müssen. Es sind außer den hier vorgestellten Typen weitere Optimierungen für den String Typ denkbar, doch wurden im Rahmen dieser Arbeit nur diese im folgenden beschriebenen Typen betrachtet. Diese befinden sich im Maven Artefakt *faststring-core*.

3.1 SubstringString

Der Typ `de.unifrankfurt.faststring.core.SubstringString` dient als Optimierung für die Methode `java.lang.String.substring(..)`. Ein `char` Array dient als Wert für die repräsentierte Zeichenkette. Zusätzlich besitzt der Typ zwei Zeiger `start` und `end`, die jeweils auf den Anfang und das Ende der repräsentierten Zeichenkette zeigen. Beim Aufruf von `substring` wird eine neue Instanz dieses Typs erzeugt, der eine Referenz auf das `char` Array, sowie die neu errechneten Zeiger auf den Anfang und das Ende der neu errechneten Zeichenkette im Konstruktor übergeben werden.

Um eine größere Kompatibilität mit der ursprünglichen String API zu erlangen, wurden zusätzliche Methoden, die einfach für die neuen Datenstruktur zu implementieren waren, zu diesem Typ hinzugefügt. Dazu gehört zum einen das `java.lang.CharSequence` Interface, zum anderen einige weitere Methoden, die gewöhnlich im Zusammenhang mit `substring` aufgerufen werden.

Zu den Methoden aus dem `CharSequence` Interface gehören:

- `length()`: Implementiert durch `ende - start`
- `charAt(i)`: Implementiert durch `value[start + i]`

- `subSeqeunce(start, end)`: Implementiert durch `substring(start, end)`

Zusätzliche Methoden werden benötigt, um eine Umwandlung von einem String zu einem `SubstringString`, sowie in die entgegengesetzte Richtung, zu ermöglichen. Dazu gehören:

- `<init>(String)`: Konstruktor, um eine neue Instanz mit dem Wert der übergebenen Zeichenkette zu erzeugen
- `String toString()`: überschriebene `java.lang.Object.toString()` Methode, um eine Umwandlung von einem `SubstringString` zu einem String zu ermöglichen
- `SubstringString valueOf(String)`: statische *Factory Methode* zum statischen erzeugen einer neuen Instanz (Wird für simplere Bytecode Generierung verwendet)

3.2 StringListBuilder

Die Konkatenation von Strings kann in Java mit dem + Operator vorgenommen werden. Allerdings wird in zahlreichen Java Coding Guidelines der `String Builder` bzw. `StringBuffer` als Alternative für den + Operator empfohlen. Grund für die Empfehlung ist, dass, wie bereits beschrieben, die Manipulationen von Strings zu dem Kopieren der Zeichen des `char` Arrays führt. Aufgrund dieser Problematik wird ein auf Strings angewandter + Operator seit Java 6 zu einem Aufruf von `StringBuilder.append()` kompiliert.

Die `StringBuilder` Implementierung basiert auf einem `char` Array, welches durch den Aufruf der `append` mit den übergebenen Werten befüllt wird. Ist dieses Feld voll, so wird ein neues Feld mit doppelter Größe angelegt und die Werte des alten Arrays in das neue kopiert.

Diesen Kopiervorgang umgeht der `de.unifrankfurt.faststring.core.StringListBuilder`, indem er eine verkettete Liste von Strings verwaltet und dieser bei jedem `append` Aufruf ein neues Element hinzufügt. Als Anker werden der Kopf sowie das Letzte Element der Liste gespeichert.

Allerdings erfüllt dieser Datentyp nicht die Schnittstelle des `java.lang.StringBuilders`. Der `StringListBuilder` erfüllt den Zweck Instanzen vom Typ `SubstringString` zu konkatenieren. Daher existiert die Methode `append(SubstringString)`, aber keine entsprechende Methode für `java.lang.String`.

Da dieser Typ weniger auf Konvertierung innerhalb des Programmflusses, als eher auf die komplette lokale Ersetzung des ursprünglichen `StringBuilders` innerhalb einer Methode abzielt, werden die Methoden zur Konvertierung wie im `SubstringString` nicht benötigt. Zusätzlich existieren die Methoden:

- `<init>()`: Default Konstruktor
- `<init>(SubstringString)`: Zum Erzeugen einer Instanz mit einer Vorbelegung.
- `toString()`: wie auch `java.lang.StringBuilder.toString()` erzeugt die Methode den erstellten String.

4 Analyse

Das folgende Kapitel beschreibt den Analyse Algorithmus, des von mir entworfenen Systems. Im ersten Abschnitt sollen die verwendeten Datenstrukturen vorgestellt und beschrieben werden. Der zweite Abschnitt beschreibt schließlich den Algorithmus.

4.1 Datenstrukturen

Für den Algorithmus wurden zwei grundlegende Datenstrukturen entworfen. Der *Datenflussgraph* repräsentiert den Datenfluss der Referenzen innerhalb einer Methode und wird im ersten Abschnitt vorgestellt. Zu optimierende Referenzen werden in diesem Graphen mit sogenannten *Labels* versehen. Dieser Datentyp soll im zweiten Abschnitt beschrieben werden.

4.1.1 Datenfluss Graph

Eine auf einem IR basierende Methode wird vom System mittels eines Datenflussgraphen repräsentiert. Dieser wird vor der Analyse aus der zu optimierenden Methode und einer Menge an initialen Referenzen vom `DataFlowGraphBuilder` erzeugt. Der Graph ist gerichtet und setzt sich aus zwei verschiedenen Knoten zusammen:

- `Reference`: eine *value number* aus dem IR
- `InstructionNode`: eine Instruktion aus dem IR

Sei im Folgenden der Datenflussgraph $G = (V, E)$, R die Menge aller `Reference` Knoten und I die Menge aller `InstructionNode`s.
Im Graph gilt $\forall(x, y) \in V, (x \in R \wedge y \in I) \vee (x \in I \wedge y \in X)$. Eine Kante $i \in I, r \in R, (i, r)$ beschreibt eine *Definition*, die aussagt, dass die Referenz r durch die Instruktion i definiert wird. Ein Kante $i \in I, r \in R, (r, i)$ ist eine *Benutzung* (im folgenden *Use* genannt).

`Reference` Knoten werden für jede betroffene *value number* erzeugt. Für die Erzeugung von `InstructionNode` Objekten steht die `InstructionNodeFactory` zur Verfügung, die für eine gegebene `SSAInstruction` eine entsprechende `InstructionNode` erstellt. Um für dieselbe `SSAInstruction` immer dasselbe `InstructionNode` Objekt zu garantieren, verwendet die Factory einen internen Cache, der eine Abbildung $SSAInstruction \rightarrow InstructionNode$ verwaltet, und für jede `SSAInstruction` prüft, ob für diese bereits eine `InstructionNode` erstellt worden ist.

Die Erstellung eines `DataFlowGraph`s beginnt immer mit einer Menge an initialen `Reference` Objekten. Ausgehend von dieser Startmenge werden über das `DefUse`-Objekt des betroffenen IRs die *Definition* und alle Uses, dieser Referenzen, in den Datenflussgraphen eingefügt. Algorithmus 1 beschreibt die Erstellung des Graphen. Die verwendete Queue Implementierung ist eine auf einem `java.util.LinkedHashSet` basierende Eigenentwicklung mit dem Namen `de.unifrankfurt.faststring.analysis.util.UniqueQueue`.

Algorithm 1 Erstellung des Datenflussgraphen

```
1:  q ← new Queue(initialReferences)
2:  g ← new DataFlowGraph()
3:  while not q.isEmpty() do
4:    r ← q.remove()
5:    if not g.contains() then
6:      def ← defUse.getDef(r)
7:      uses ← defUse.getUses(r)
8:      newInd.add(def)
9:      newInd.add(uses)
10:     r.setDef(factory.create(def))
11:     for ins ∈ defUse.getUses(r) do
12:       r.addUse(factory.create(ins))
13:     end for
14:     for ins ∈ newIns do
15:       q.add(ins.getConnectedRefs())
16:     end for
17:     g.add(r))
18:   end if
19: end while
20: return g
```

Jede `InstructionNode` besitzt eine *Definition*, die *value number*s der Referenz die diese Instruktion erzeugt, und eine Liste von *Use*s, die *value number*s der Referenzen die es benutzt. Darüber hinaus besitzt es noch Informationen zu Bytecode Spezifika, die im Kapitel 5.1 betrachtet werden.

Für verschiedene `SSAInstruction` Typen existieren entsprechende `InstructionNode` Subtypen. Allerdings können nicht alle Typen einer `SSAInstruction` zugeordnet werden. Im Folgenden sollen die wichtigsten Knotentypen vorgestellt werden. Es existieren darüber hinaus weitere Typen von Instruktionen, für die das System zur Zeit keine Unterstützung bietet, da es ausschließlich für komplexe Objekte ausgelegt ist.

Einen weiteren Subtyp bildet die `LabelableNode`. Dieser Typ stellt eine speziellen Knoten dar, der mit einem Label markiert und damit auch optimiert werden kann.

MethodCallNode

Eine `MethodCallNode` repräsentiert einen Methoden Aufruf. Es besitzt, wenn vorhanden, eine *Definition* (nur vorhanden, bei Zuweisung des Rückgabewert), einen Receiver, wenn es keine statische Methode ist, eine Liste an Parametern und die aufgerufene Methode. Dieser Typ ist eine `LabelableNode`.

ContantNode

Dieser Knoten Typ stellt eine Konstanten *Definition* dar. Er besitzt ausschließlich die *Definition*, welcher Referenz diese Konstante zugewiesen wird. Für den Typ existiert keine entsprechende `SSAInstruction`.

ParameterNode

Die `ParameterNode` wird als *Definition* der Parametern der Methode verwendet. Wird eine Variable innerhalb der Methode als Parameter in der Methoden Signatur deklariert, wird deren *Definition* als `ParameterNode` im Datenflussgraphen repräsentiert. Für diesen Typ existiert keine entsprechende `SSAInstruction`.

NewNode

Dieser Typ entspricht einer `NEW` Anweisung, die ein neues Objekt eines gegebenen Typen erstellt. Es besitzt eine *Definition* und den Typ des instanziierten

Objekts. Die *NewNode* ist eine `LabelableNode`.

ReturnNode

`ReturnNode` Typen sind `RETURN` Anweisungen. Sie besitzen (in Java) ausschließlich eine Referenz als *Use* und zwar diejenige Referenz, die sie aus der Methode zurückgeben. Dieser Typ kann keine *Definition* darstellen.

PhiNode

Die `PhiNode` steht für eine ϕ-Instruktion aus dem IR. Sie verfügt über eine Referenz als *Definition* und 2 bis n *Use*s. Dieser Typ ist eine `LabelableNode`.

4.1.2 Label

Das *Label* entspricht einer Markierung, mit der Knoten in einem Datenflussgraphen versehen werden können. Dabei steht ein Label (oder `TypeLabel`, wie der Datentyp im System heißt) für einen optimierten Typ. Die Semantik hinter einem markierten Knoten ist, dass diese Referenz bzw. Instruktion durch den entsprechenden optimierten Typ ersetzt werden kann.

Es kann nicht für alle `InstructionNode`s ein Label gesetzt werden. Genauer gesagt, kann nur die Knotentypen `MethodCallNode`, `NewNode` und `PhiNode` ein Label gesetzt werden, da sich nur diese Instruktionen in einen optimierte Variante umwandeln lassen.

Das `TypeLabel` beinhaltet alle Definitionen und Regeln, die für die Verwendung eines optimierten Typen existieren. Dazu gehören

- der originale, sowie der optimierte Typ
- die Methoden, für die der optimierte Typ Optimierungen implementiert
- alle Methoden, die darüber hinaus vom Optimierten Typ unterstützt werden
- Methoden, die den optimierten Typ als Rückgabewert zurückgeben
- zu dem beschriebenen Label kompatible Label

Diese Aspekte werden durch die Methoden des Interface abgebildet. Im System lassen sich Label Definitionen auf zwei Arten erstellen:

1. Durch das Implementieren des Interfaces `TypeLabel`
2. Durch das Erstellen einer `.type` Datei

Zwar unterstützt das Kommandozeilen Tool zur Zeit nur die zweite Variante, programmatisch lässt sich allerdings auch die erste umsetzen. Im Folgenden sollen die beiden Möglichkeiten zur Definition eines `TypeLabel`s betrachtet werden.

Das TypeLabel Interface

Das Interface beinhaltet alle Methoden, die der Analyse- und Transformationsprozess benötigt. In diesem Kapitel sollen zunächst nur die Methode betrachtet werden, die für den Analyse Algorithmus verwendet werden, die übrigen werden im Abschnitt 5.2.1 erläutert.

`canBeUsedAsReceiverFor(MethodReference):boolean` - Legt fest, ob eine markierte Referenz als Empfänger für den übergebenen Methodenaufruf dienen kann.

`canBeUsedAsParamFor(MethodReference,int):boolean` - Bestimmt, ob eine markierte Referenz als Parameter in dem gegebenen Methodenaufruf an der entsprechenden Stelle (der `int` Parameter) verwendet werden kann.

`canBeDefinedAsResultOf(MethodReference):boolean` - Sagt aus, ob die gegebene Methode einen optimierten Typ zurückgeben kann. Dies impliziert, dass der Methodenaufruf selber auch markiert ist.

`findTypeUses(AnalyzedMethod):Set<Reference>` - Gibt eine Menge an `Reference` Objekten zurück, auf denen innerhalb der gegebenen Methode eine der von der Optimierung betroffenen Methode aufgerufen wird. Für diesen Algorithmus existiert bereits eine Implementierung in der Klasse `BaseTypeLabel`.

`compatibleWith(TypeLabel):boolean` - Gibt an, ob das übergebene Label kompatibel mit diesem Label ist.

All diese Methoden werden von `InstructionNode` Implementierungen verwendet, um Aussagen über die Label Konformität zu treffen. Wie das detailliert geschieht, wird im Abschnitt *Algorithmus* beschrieben.

Das .type Dateiformat

Da das Implementieren des Interfaces eher komplex ist, wurde für die einfachere Definition eines Types ein Dateiformat entwickelt, welches von der Komplexität des Interfaces abstrahieren soll. In dieser werden nicht die Regeln selbst, sondern die Fakten beschrieben, aus denen die Regeln für den Algorithmus hergeleitet werden können.

Aus einer Datei im `type` Format wird mittels eines internen Parsers ein `TypeLabelConfig` Objekt erzeugt, welches als `TypeLabel` Objekt für den Algorithmus dient.

Für die inhaltliche Struktur der Datei wurde die JSON (JavaScript Object Notation) Syntax gewählt, um eine Darstellung anzubieten, die sowohl für Menschen als auch für das Programm leicht zu lesen und zu verstehen ist. Die Attribute innerhalb dieser Datei werden nachfolgend erläutert. Abbildung 4.1 zeigt ein Beispiel einer `type`-Datei.

name Name des Labels

originalType voll qualifizierte Name des zu ersetzenden Typs

optimizedType voll qualifizierte Name des zu optimierten Typs

methodDefs Liste von Methoden Definitionen. Jede in der Definition verwendeten Methode muss in dieser Liste mit einer ID versehen werden. Diese Methode lässt sich über diese ID referenzieren und lässt sich nur über diese ID verwenden. Ein Eintrag ist ein Objekt mit den Attributen:

- **id** eindeutige ID für die diese Methode
- **desc** Beschreibung dieser Methode. Dies ist ein weiteres Objekt und besteht aus den Attributen:
  - **name** Name der Methode
  - **signature** Signatur der Methode zusammengesetzt aus der Parameterliste und den Rückgabewert. Die Typen müssen dabei in der internen JVM Form angegeben werden. (Beispiel: `"(I)Ljava/lang/String;"`, ein Parameter vom Typ `int` und Rückgabewert vom Typ `java.lang.String`)

effectedMethods Liste von Methoden IDs. Für diese Methoden existieren optimierte Varianten in dem optimierten Typen.

supportedMethods Liste von Methoden IDs. Diese Methoden werden vom optimierten Typ zusätzlich unterstützt. Bei den hier angegebenen Methoden handelt es sich nicht um Optimierungen.

producingMethods Liste von Methoden IDs. Diese Methoden erzeugen in ihrer optimierten Variante optimierte Typen.

compatibleLabel Liste von Strings. Beinhaltet alle Label, die mit diesem Label kompatibel sind.

parameterUsage Ein Objekt. Dabei ist jeder key die ID einer Methode und der entsprechende value eine Liste von Ganzzahlen. Ein Eintrag bedeutet, dass diese Methode einen optimierten Typ als Parameter mit diesem Index erwartet.

optimizedParams Ein Objekt. Dabei ist jeder key die ID einer Methode und der entsprechende value eine Parameter Liste in interner JVM Notation. Wird bei einer Optimierung einer Methode eine andere Parameter Liste erwartet als die ursprüngliche der originalen Methode, so muss die neue Signatur an dieser Stelle angegeben werden.

staticFactory Ein String. Der Name einer statischen Factory Methode mit einem Übergabeparameter vom Typ des originalen Typs. Diese muss einen entsprechenden optimierten Typ zurückgeben.

toOriginalType Ein String, Der Name einer Methode ohne Parameter, die aus dem optimierten Objekt, ein entsprechendes vom originalen Typ zurückgibt.

```
{
    "name" : "SubstringString",
    "originalType" : "java.lang.String",
    "optimizedType" :
        "de.unifrankfurt.faststring.core.SubstringString",
    "methodsDefs" : [
        {
          "id" : "substring_i",
          "desc" : { "name" : "substring",
                     "signature" : "(I)Ljava/lang/String;" }
        },
        {
          "id" : "substring_ii",
          "desc" : { "name" : "substring",
                     "signature" : "(II)Ljava/lang/String;" }
        },
        {
          "id" : "length",
          "desc" : { "name" : "length", "signature" : "()I" }
        },
        {
          "id" : "charAt",
          "desc" : { "name" : "charAt", "signature" : "(I)C" }
             },
        {
          "id" : "valueOf",
          "desc" : { "name" : "valueOf",
                     "signature" :
                     "(Ljava/lang/Object;)Ljava/lang/String;" }
        }
    ],

    "effectedMethods" : ["substring_i", "substring_ii"],
    "supportedMethods" : ["length", "charAt"],
    "producingMethods" : ["substring_i", "substring_ii"],

    "compatibleLabels" : ["StringListBuilder"],

    "staticFactory" : "valueOf",
    "toOriginalMethod" : "toString"
}
```

Abbildung 4.1: Beispiel einer *type* Datei, anhand der SubstringString Definition

4.1.3 Konventionen

Bei dem Erstellen von optimierten Typen sind einige Konventionen zu befolgen. Das System rechnet damit, dass diese Annahmen befolgt werden.

Methodennamen

Wird eine Methode als *effectedMethod* deklariert, wird der optimierte Gegensatz durch den Namen identifiziert. Wenn z.B. die Methode `f()` aus dem Typ `A` optimiert werden soll, so muss in dem optimierten Typ `AOpt` eine Methode `f()` existieren. Besitzt die originale Methode eine Parameterliste, so muss diese mit derer der optimierten Methode in Anzahl und Typen übereinstimmen. Eine Ausnahme bildet dabei die Verwendung von ebenfalls optimierten Parametern. In diesem Fall muss im Feld *optimizedParams* in der *type* Datei ein entsprechender Eintrag erfolgen.

4.2 Algorithmus

In diesem Abschnitt soll die Idee hinter dem Analyse Algorithmus sowie dessen Implementierung vorgestellt werden. Hierzu wird zunächst das Konzept der 'Bubble' erläutert, um danach auf diesem Konzept aufbauenden Algorithmus zu betrachten.

4.2.1 Motivation

Optimierte Referenzen sind im Falle von String Objekten nicht kompatibel mit der originalen Implementierung. So treten Probleme in den folgenden Szenarien auf:

Referenz als Methodenparameter Die Signatur der optimierten Methode, wird nicht verändert, da es dazu führen würde, dass Klienten Code, der diese Methode weiterhin mit dem originalen Typ aufruft, nicht mehr kompilieren würde.

Referenz als Rückgabewert Ein ähnliches Problem existiert, wenn die Referenz zurückgegeben wird. Die Signatur der Methode definiert den originalen Typ als Rückgabetyp und das Zurückgeben eines anderen Typs als der definierte würde zu Laufzeitfehlern führen.

Methodenaufruf auf Referenz Wird diese optimierte Referenz als Empfänger von einer Methode verwendet, die nicht zu den optimierten oder unterstützen Methoden dieses optimierten Typs gehört, würde es zu Laufzeitfehlern kommen, da diese aufzurufende Methode nicht im optimierten Typ vorhanden ist.

Feld Zugriff (`GETFIELD`, `PUTFIELD`) Wird die optimierte Referenz in ein oder aus einem Feld innerhalb eines Objektes (oder in ein statisches Feld einer Klasse) gesetzt, stimmt auch in diesem Szenario der Typ des optimierten und des originalen Objekts nicht überein.

Array Zugriff Bei dem Zugriff auf ein Array, sowohl lesend als auch schreibend, existiert eine Unstimmigkeit mit dem Typ des Arrays.

Referenz Aufruf Parameter Wird die Referenz als Parameter für einen Methoden Aufruf verwendet, die nach Label Definition nicht mit einem optimierten Typ kompatibel ist, dann erwartet diese Methode den originalen Typ und nicht den optimierten.

Erweitert der optimierte Typ seinen originalen Typ, so wäre durch den Polymorphismus der optimierte Typ genauso wie der originale verwendbar. Allerdings ist der Typ `java.lang.String` final, was bedeutet, dass von diesem Typ nicht abgeleitet werden kann. Darüber hinaus bieten sich Ableitungen für Optimierungen nicht an, da das dynamische Dispatchen zusätzlichen Aufwand für die JVM darstellt. Die Begründung liegt darin, dass dabei zunächst die Implementierung der Methode zu lokalisieren ist, bevor der Methodenaufruf auf nicht finalen Methoden durchgeführt werden kann.

Aus diesem Grund müssen in den Code Konvertierungen zwischen originalem und optimiertem Typ eingefügt werden. Eine Konvertierung zum optimierten Typ muss vor dem zu optimierenden Methodenaufruf erfolgen. Eine Umwandlung vom optimierten Typ muss vor einer nicht kompatiblen Benutzung dieser Referenz erfolgen um bei dieser den originalen Typ zu verwenden.

4.2.2 Die Bubble

Da Konvertierungen zusätzliche Laufzeit erfordern, muss es das Ziel sein, die Anzahl der durchgeführten Konvertierungen zu minimieren. Dies wird erreicht, indem man den Bereich, in dem ein optimierter Typ statt des originalen innerhalb der Methode verwendet wird, maximiert. Dieser Bereich, in dem ein

optimierter, statt des originalen Typs für eine Referenz verwendet wird, wird im nachfolgend *Bubble* genannt.

Die Bubble entsteht mittels der Markierung von Knoten im Datenflussgraphen. Es wird für jede gegebene Instanz vom Typ `TypeLabel` eine Bubble auf dem Graphen erzeugt. Dabei kann ein Knoten mit maximal einem Label markiert sein, daher kann ein Knoten immer nur Teil einer Bubble sein.

Ziel des Algorithmus ist es, diese Bubble ausgehend von den zu optimierenden Methoden so weit wie möglich zu erweitern. Als Anfangszustand werden alle zu optimierenden Methodenaufrufe mit dem zu verarbeitenden Label markiert, die alle für sich einzelne Bubbles darstellen.

4.2.3 Umsetzung des Algorithmus

Als Eingabe für den Algorithmus dient ein Objekt vom Typ `DataFlowGraph`, der den bereits beschriebenen Datenflussgraphen darstellt. Zum Erstellen des Graphen werden wie in 4.1.1 beschrieben zunächst initiale Referenzen benötigt. Diese Referenzen werden über die Methode `Set<Reference> findTypeUses(AnalyzedMethod)` ermittelt, welche sich in der abstrakten Klasse `BaseTypeLabel` befindet. Diese Methode ist auch Teil des `TypeLabel` Interfaces. Die betreffenden Methoden werden mittels der in der Label Definition definierten *effectedMethods* identifiziert, also derjenigen Methoden, für die dieses Label eine Optimierung darstellt. Die `findTypeUses` Methode erstellt für jede Referenz, auf der in der gegebenen Methode eine der *effectedMethods* aufgerufen wird, ein `Reference` Objekt. Dabei wird für jede *value number* genau ein Referenz Objekt erzeugt. Die erstellten Referenzen werden dem verwendeten Label markiert.

Die Implementierung des Algorithmus befindet sich als statische Methode in der Klasse `LabelAnalyzer`. Ausgehend von den initial markierten Knoten, werden jeweils die *Definition* und *Use*s einer Referenz betrachtet, ob diese mit dem Label der Referenz markiert werden können. Der Algorithmus 2 beschreibt die Implementierung der Vererbung des Labels. Als Queue findet die `de.unifrankfurt.faststring.analysis.util.UniqueQueue` Verwendung.

Algorithm 2 Vererbung des Labels

```
1: q ← new Queue(initialReferences)
2: g ← der übergebene Datenflussgraph
3: while not q.isEmpty() do
4:   r ← q.remove()
5:   if r.label is not null then
6:     def ← r.getDef()
7:     if def.canProduce(r.label) then
8:       labelConnectedRefs(def)
9:     end if
10:    for use in r.getUses() do
11:      if use.canUseAt(r.label, useIndex) then
12:        labelConnectedRefs(use)
13:      end if
14:    end for
15:  end if
16: end while
17: return g
```

Eine wichtige Rolle während des Vererbungsprozesses spielen dabei die Methoden `boolean Labelable.canProduce(TypeLabel)` und `boolean Labelable.canUseAt(Ty int)`. Diese beiden Methoden bestimmen für eine Referenz, ob ein Label auf eine *Definition* oder ein *Use* vererbt werden kann.

`canProduce` sagt aus, ob eine Instruktion eine optimierte Referenz definieren kann. Die Implementierungen der `canProduce` Methode sehen für die einzelnen `Labelable` Subtypen wie folgt aus:

NewNode gibt immer `true` zurück.

ConditionalBranchNode gibt immer `false` zurück.

MethodCallNode delegiert den Aufruf an die Methode `canBeDefinedAsResultOf` der übergebenen Label Instanz.

Die Methode `canUseAt` betrifft dagegen Verwendungen von Referenzen. Die Methode beantwortet die Frage, ob diese Instruktion eine optimierte Referenz verwenden kann. Der übergebene Integer Wert stellt dabei den Usage Index

dar, an dem diese Referenz in dieser Instruktion verwendet wird. Die Implementierung dieser Methode sieht für die einzelnen `Labelable` Subtypen wie folgt aus:

NewNode gibt immer `false` zurück.

ConditionalBranchNode gibt immer `true` zurück.

MethodCallNode Wenn die Methode nicht statisch und der Index = 0 ist, wird an die Methode `canBeUsedAsReceiverFor` delegiert, andernfalls an die Methode `canBeUsedAsParamFor`.

Ist für eine Instruktion die Prüfung für die entsprechende Methode positiv, wird diese Instruktion an die Hilfsfunktion `labelConnectedRefs` weitergeleitet.

Algorithm 3 labelConnectedRefs

```
1: node.setLabel(label)
2: for ref in node.getLabelableRefs() do
3:   if ref.getLabel() is not null then
4:     ref.setLabel(label)
5:     q.add(ref)
6:   end if
7: end for
```

Wobei sowohl die Queue q, als auch das betreffende TypeLabel *label* aus dem umgebenen Kontext im Algorithmus 2 in dieser Funktion zur Verfügung stehen.

4.2.4 Umgang mit mehreren Labels

Es ist möglich für die Analyse mehrere Labels für die Analyse zu verwenden. Diese lassen sich dem `MethodAnalyzer` als `Collection` im Konstruktor zusammen mit der zu analysierenden Methode übergeben. Vor der Analyse werden zuerst alle Referenzen identifiziert, von denen eine zu optimierende Methode verwendet wird. Diese Referenzen werden zu einer Menge an Referenzen vereint. Die auf diese Weise erzeugten Referenzen besitzen, wie in 4.2.3 beschrieben, jeweils das Label, für das die Referenz erzeugt wurde.

Stellt man der Analyse mehrere Labels zur Verfügung, die allerdings Optimierungen für dieselbe Methoden desselben Typs beschreiben, ergibt sich ein Problem. Da jede Referenz immer nur mit einem Label markiert werden kann, werden diejenigen Referenzen, welche dieselbe zu optimierende Methode verwenden, mit demjenigen Label markiert, welches zuerst verarbeitet wird. Das wird immer das Label sein, welches einen geringeren Index in der Liste von übergebenen Labels besitzt.

Stellt man der Methode nun zwei unterschiedlichen Labels zur Verfügung, die denselben originalen Typen verwenden, ergibt sich ein anderes Problem, und zwar wenn innerhalb der analysierten Methode auf einer Referenz diesen Typs sowohl die eine als auch die andere zu optimierende Methode aufgerufen wird. Da jeder Referenz in einer Methode maximal ein Label zugewiesen wird, kann die betroffene Referenz, welche beide Methoden der beiden TypeLabels verwendet, nur mit einem der TypeLabels markiert werden. In diesem Szenario wird ebenfalls dasjenige Label für die Markierung verwendet, welches den niedrigeren Index innerhalb der Liste besitzt.

Nachdem der Datenflussgraph initial erzeugt wurde, wird der Algorithmus wie in 4.2.3 beschrieben ausgeführt. Da Labels nur auf Knoten vererbt werden, gilt es dass der entsprechende Knoten dasjenige Label erhält, welches zuerst auf diesen Knoten vererbt wird.

4.2.5 Phi-Knoten

ϕ-Instruktionen innerhalb des Datenflussgraphen erfordern aufgrund ihrer Eigenheiten eine besondere Behandlung. Da sie keine Instruktionen im eigentlichen Sinne darstellen, sondern nur ein Zeiger Alias für andere Referenzen sind, lassen sich diese Knoten nicht optimieren. Obwohl sie selber nicht optimiert werden können, lässt sich doch über diese Knoten hinweg ein Label vererben. Allerdings sollte unterschieden werden zwischen denjenigen Referenzen innerhalb Instruktion, für die eine Markierung tatsächlich sinnvoll und für welche eher weniger sinnvoll ist. Es sollten diejenigen Referenzen markiert werden, für die auch Optimierungspotenzial besteht. Wohingegen die Referenzen, für die keinerlei Optimierung notwendig ist, keine Markierung erhalten sollten.

Wenn das System beim Vererben der Markierung einen ϕ-Knoten als Instruktion verarbeitet, wird dieser auf den ϕ Stapel gelegt und alle verbundenen Referenzen für die weitere Verarbeitung vorgemerkt. Nachdem alle Referenzen verarbeitet worden sind, wird dieser Stapel durchlaufen. Für jede `PhiInstructionNode` in diesem Stapel wird entschieden, ob und mit welchem

Label diese Instruktion markiert wird. Diese Entscheidung wird über die Häufigkeit einer Label-Markierung in der Menge aller verbundenen Referenzen getroffen.

Seien $L = \{l_1, ..., l_n\}$ alle Labels der aktuellen Analyse, inklusive des Labels für keine Optimierung, in ihrer natürlichen Reihenfolge und die Funktion $anzahl : L \to \mathbb{N}$ die Anzahl der Markierungen aller verbunden Referenzen. Dann ist $m = max(\{anzahl(l_i)|l_i \in L\})$ die Markierung für den verarbeiteten ϕ-Knoten, wenn m eindeutig ist. Ist das Maximum nicht eindeutig, so wird aus der Menge der Labels mit der größten Häufigkeit, dasjenige gewählt, welches den geringsten Index besitzt.

5 Transformation

In diesem Kapitel sollen die Überlegungen und der Prozess der Bytecode Transformation auf Basis der Resultate aus dem Analyse Prozess vorgestellt werden. Es wird zunächst auf die Beschaffung der nötigen Informationen eingegangen, um im Anschluss die Regeln, nach denen Bytecode generiert oder manipuliert wird, zu erläutert.

Ziel der Transformation ist es zum einen an den Grenzen der Bubble Konvertierungen zwischen den originalen und den optimierten Typen in den Sourcecode einzufügen. Zum anderen müssen markierte *Use*s in entsprechende optimierte Versionen umgewandelt werden.

5.1 Lokale Variablen

5.1.1 Optimierte Variablen

Um originale lokale Variablen im Bytecode nicht mit den optimierten Versionen zu überschreiben, wurde eine Abbildung geschaffen, die jeder lokalen Variable ein Tupel zuweist: $l \rightarrow (L, l')$, wobei l die originale lokale, L ein Label und l' die optimierte Variable für das Label L darstellt. So ist sichergestellt, dass optimierte und die entsprechende originale Referenz in zwei verschiedenen Lokalen geführt werden. Darüber hinaus ist diese Trennung wichtig, da die JVM lokale Variablen typisiert, und dadurch nicht an verschiedenen Stellen der Methode unterschiedliche Typen in derselben lokalen Variable gespeichert werden können.

5.1.2 Variablen zu Value Numbers

Da der IR mit den beinhalteten *value number*s eine Abstraktion des eigentlichen Bytecodes darstellt, fehlt auch jeglicher Bezug zu den eigentlichen lokalen Variablen, die von einer spezifischen *value number* dargestellt werden. Zusätzlich muss für die *Definition* einer Instruktion das `STORE` (schreibt die auf dem

Stack liegende Referenz in die gegebene lokale Variable) und für alle *Use*s entsprechende `LOAD`s (liest die gegebene lokale Variable) im Bytecode lokalisiert werden. Diese Informationen sind nötig, da im Falle von Optimierungen, die für die entsprechende Instruktion erzeugt werden, die optimierten statt die originalen Referenzen geladen werden müssen.

Diese Informationen werden in der `InstructionNode` gehalten. Beim Erzeugen eines solchen Objekts wird in der `InstructionNodeFactory` zum einen versucht die lokalen Variablen zu den verwendeten *value number*s zu erschließen, und zum anderen auf die Position im Bytecode zu schließen, an der die entsprechenden Werte auf den Stack gelegt werden.

Der IR, der aus einer class-Datei erzeugt wird, besitzt ein privates Feld `localMap` vom Typ `com.ibm.wala.ssa.IR.SSA2LocalMap`, welches in ihrer einzigen Implementierung (der `com.ibm.wala.ssa.SSABuilder.SSA2LocalMap`) eine private Methode mit Signatur `int[] findLocalsForValueNumber(int, int)` besitzt. Diese Methode gibt für eine gegebenen Bytecode Index und *value number* alle möglichen lokalen Variablen für diese *value number* an der gegebenen Stelle zurück. Um diese Methode trotz aller Zugriffsbeschränkungen aufzurufen, wurde eine Methode in *Groovy* geschrieben. Beim Suchen nach lokalen Variablen muss zwischen *value number*s als *Definition* und als *Use* unterschieden werden. Das folgende Beispiel beschreibt das Problem bei *Definitionen*:

```
INVOKEVIRTUAL org/example/SomeType.f()I // index 1
ISTORE 5                                // index 2
```

Abbildung 5.1: Lokale Variable für Definition

Die lokale Variable der *Definition* der `INVOKEVIRTUAL` Instruktion ist zum Zeitpunkt des Methodenaufrufs noch nicht bekannt. Erst im Index 2 wird dieser Wert der lokalen Variablen 5 zugewiesen.

Um nun die Stellen zu finden, an denen Variablen auf den Stack gelegt oder vom Stack gepoppt werden, wurde eine einfache Stacksimulation eingeführt, wie sie in Algorithmus 4 zu sehen ist.

Algorithm 4 Simulation des Stacks

```
1: size ← Höhe des Stacks zum Zeitpunkt der Instruktion
2: index ← Index der betroffenen Referenz innerhalb des Stacks
3: bcIndex ← Bytecode Index der betroffenen Instruktion
4: while actBlock.getPredNodes() = 1 do
5:   actBlock ← callGraph.getBlockFor(bcIndex)
6:   while bcIndex > actBlock.getFirstInstructionIndex() do
7:     bcIndex ← bcIndex − 1
8:     instruction ← instructions[bcIndex]
9:     if instruction.getPushedCount() > 0 then
10:      size ← size − 1
11:      if index == size then
12:        return bcIndex
13:      end if
14:    end if
15:    size ← size + instruction.getPoppedCount()
16:  end while
17: end while
18: return −1 // kein Index gefunden
```

Dieser Algorithmus funktioniert für *Definitionen*, also das Suchen von `STORE` Instruktionen, ähnlich. Der Unterschied liegt dabei ausschließlich im Inkrementieren (statt Dekrementieren) der *bcIndex* Variable und dem umgekehrten Verhalten beim *push* bzw. *pop* von Werten auf bzw. vom Stack.

Die Informationen werden in dem entsprechenden `InstructionNode` Objekt gespeichert. Zu diesem Zweck besitzt dieser Typ drei Abbildungen ($\mathbb{N} \to \mathbb{N}$), die zum Zeitpunkt der Erstellung befüllt werden:

`localMap` bildet eine *value number* auf eine lokale Variable ab.

`loadMap` bildet eine lokale Variable auf einen Bytecode Index ab, an dem diese Variable auf den Stack geladen wurde.

`storeMap` bildet eine lokale Variable auf einen Bytecode Index ab, an dem `STORE` die Referenz in die entsprechende Variable schreibt.

5.2 Bytecode Generierung

Die Generierung von neuem Bytecode und die Manipulation von bestehendem wird von einer Instanz der Klasse `MethodTransformer` vorgenommen. Als Eingabe dient eine Instanz vom Typ `TransformationInfo`, welche auf einem `AnalysisResult` basiert, und Informationen über die verwendeten lokalen Variablen zur Verfügung stellt. Die Informationen sind die verwendeten Variablen und die Abbildung von originalen Lokalen zu optimierten.

Anhand der `TransformationInfo` werden sowohl Konvertierungen wie auch Optimierungen zu dem bestehenden Bytecode hinzugefügt. Eine Konvertierung beschreibt dabei eine Transformation von einem originalen zu einem optimierten Typ bzw. die von einem optimierten Typ zu einem originalen Typ. Eine Optimierung hingegen ersetzt eine Instruktion mit einer optimierten Variante. Diese Mechanismen sollen in dem folgenden Abschnitt vorgestellt werden.

5.2.1 Informationen des TypeLabels

Um Optimierungen und Konvertierungen auf bestimmten Typen umzusetzen werden verschiedene Informationen benötigt. Diese setzen sich zusammen aus:

- Wie der optimierte Type heißt,
- Wie ein optimierter Typ aus einem originalen erzeugt wird,
- Wie die Signatur einer optimierten Variante einer Methode aussieht.

Diese Informationen werden über die `TypeLabel` *Definitionen* dem System zur Verfügung gestellt. Zu diesem Zweck enthält das `TypeLabel` Interface, zusätzlich zu denen in 4.1.2 vorgestellten, die folgenden Methoden:

`getOptimizedType():Class<?>` gibt ein `Class` Objekt des optimierten Typs zurück

`getOriginalType():Class<?>` gibt ein `Class` Objekt des originalen Typs zurück

`getCreationMethodName():String` definiert den Namen der statischen Methode innerhalb des optimierten Typs, die ein neues Objekt des optimierten Typs erzeugt.

`getToOriginalMethodName():String` definiert den Namen einer Methode, die auf einem Objekt des optimierten Typen aufgerufen werden kann, um eine äquivalente Instanz des originalen Typen zu erzeugen.

`getReturnType(MethodReference):Class<?>` gibt den Rückgabe Typ der gegebenen Methode im optimierten Typ zurück. Der Typ wird als `String` Repräsentation in der internen JVM Form zurückgegeben. (Beispiel: `Ljava/lang/String;`

`getParams(MethodReference:String` gibt die Parameter der gegebenen Methode im optimierten Typ zurück. Die Parameter werden konkateniert und umgeben mit einfachen Klammern beschrieben. Die Typen der Parameter werden als interne JVM Form angegeben (Beispiel: `(II)`).

5.2.2 Konvertierung

Konvertierungen dienen der Kompatibilität zwischen Code innerhalb und außerhalb der "Bubble". Dabei betreffen diese ausschließlich Referenzen im Datenflussgraphen. Formal lässt sich dieser Vorgang wie folgt beschreiben. Die Funktion $label : E \rightarrow L$, wobei L die Menge aller in dem System vorhandenen Label ist. $E = R \cup I$ weist jedem Knoten eine Label Markierung zu. Ein Knoten ohne Markierung bekommt das Label $KEIN\_LABEL$ zugewiesen. Bei der Entscheidung, ob zwischen einer Instruktion i und einer Referenz r eine Konvertierung nötig ist, sind zwei Entscheidungen zu treffen:

1. Handelt es sich bei der Instruktion um eine *labelable* Instruktion.
2. Ist die Kante eine *Definition* ($i \rightarrow r$) - oder eine *Use* ($r \rightarrow i$) Kante innerhalb des Graphen.

Kann die betrachtete Instruktion ohnehin nicht mit einem Label markiert werden, so muss nur die Referenz betrachtet werden, und wenn $label(r) \neq KEIN\_LABEL$ gilt, dann muss eine Konvertierung an dieser Kante eingeführt werden.

Handelt es sich bei der Instruktion allerdings um eine *labelable* Instruktion, so trifft die Methode `boolean needsConversationTo(Label)` (s. Algorithmus 5) des Objekts, auf den die Kante zeigt, die Entscheidung, ob eine Konvertierung nötig ist.

Algorithm 5 needsConversationTo(Label)

```
1: if isSameLabel(label) then
2:   return false
3: end if
4: if label ≠ null then
5:   return not label.compatibleWith(this.label)
6: else
7:   return not this.label.compatibleWith(null)
8: end if
```

Konvertierungen werden von Instanzen vom Typ `Converter` durchgeführt. Dabei wird zwischen *Definition*s- und *Use*-Konvertierungen unterschieden, für die jeweils eigene Implementierungen der `Converter` Schnittstelle existieren. Der `Converter` nutzt das *Visitor* Muster, welches für die einzelnen Instruktionsknoten implementiert wurde. Die jeweiligen *visit*-Methoden erstellen auf der zur Verfügung gestellten `ConversionPatchFactory` die Shrike Patch Objekte, um die Bytecode Manipulationen durchzuführen.

Die `ConversionPatchFactory` dient einerseits der Erstellung von Shrike Patch Objekten und andererseits dem Hinzufügen dieser in den Bytecode der aktuell verarbeiteten Methode. Zusätzlich muss bei der Instanziierung eines Objektes angegeben werden für welche Konvertierung diese Factory verwendet wird. Zu diesem Zweck verwendet der Typ zwei Attribute vom Typ `TypeLabel`:

`from` - Der Typ den ein Objekt vor der Konvertierung besitzt.

`to` - Der Typ in den ein Objekt konvertiert werden soll.

Die Methoden, die zum Erzeugen eines Patches zur Verfügung stehen, sind die folgenden:

`createConversationAtStart(local)` erstellt eine Konvertierung am Anfang der Methode für die gegebene lokale Variable.

`createConversationAfter(local, bcIndex)` erstellt eine Konvertierung nach dem gegebenen Bytecode Index für die gegebene lokale Variable. Die Variablen Angabe kann auch weggelassen werden, dann wird der konvertierte Wert nicht in die entsprechende optimierte Variable gespeichert.

`createConversationBefore(local, bcIndex)` erstellt eine Konvertierung vor dem gegebenen Bytecode Index für die gegebene lokale Variable. Die Variablen Angabe kann auch weggelassen werden, dann wird der konvertierte Wert nicht in die entsprechende optimierte Variable gespeichert.

Soll eine Konvertierung für eine *Definition* ohne eine gegebene lokale Variable durchgeführt werden, so wird ausschließlich ein Methodenaufruf zur Konvertierung zum bzw. vom optimierten Typ aufgerufen. Wird eine lokale Variable angegeben, so wird zunächst die Referenz auf dem Stack mittels der `DUP` Anweisung verdoppelt. Nach dem Aufruf der Konvertierungs-Methode wird die optimierte Referenz in die entsprechende optimierte Variable gespeichert. Konvertierungen, die am Anfang der Methode durchgeführt werden (was ausschließlich Konvertierungen der Übergabeparameter betrifft), verhalten sich genauso wie Konvertierungen denen eine Variable mitgegeben wurde. Der Unterschied besteht darin, dass anstatt der `DUP` Anweisung ein `LOAD` für die entsprechende Variable verwendet wird, damit die Referenz für den Methodenaufruf verwendet wird.

Wird eine Konvertierung für einen *Use* benötigt, bevor die originale Referenz verwendet wird, muss sie in den originalen Type umgewandelt werden. Daher wird an der Stelle, nachdem die betroffene Referenz auf den Stack gelegt worden ist der Methodenaufruf zum Umwandeln in den originalen Typ eingefügt. Ist die lokale Variable allerdings bekannt, wird die `LOAD` Instruktion, die die originale Variable lädt durch das Laden der optimierten Variable ersetzt, um diese danach in einen originalen Typ zu konvertieren.

5.2.3 Optimierung

Für jede *labelable* Node wird, wenn sie mit einem Label markiert ist, eine Optimierung durchgeführt. Dafür sind drei Schritte notwendig. Primäres Ziel der Optimierungstransformation ist es, die originalen Methodenaufrufe durch die Optimierten zu ersetzen. Wie in 4.1.3 beschrieben, wird für eine Methoden Optimierung immer derselbe Name angenommen. In jedem Fall wird zunächst der Typ des Empfänger-Objekts auf den des optimierten Typ gesetzt. Ist für die Parameter Liste im optimierten Typ keine Alternative angegeben, wird diejenige der originalen Methode verwendet. Der Rückgabe Typ wird ebenfalls durch das *Typelabel* bestimmt. So wird das Ziel der `INVOKE` Instruktion auf den optimierten Typ gesetzt.

Zum Aufruf der Methode auf der optimierten Referenz, muss diese anstelle der originalen Referenz, zum Zeitpunkt des Methoden Aufrufs auf dem Stack vorhanden sein. Darüber hinaus müssen, wenn es sich bei den Parametern um gelabelte Referenzen handelt, auch die optimierten Parameter Referenzen statt der originalen auf dem Stack liegen. Daher müssen für optimierte Referenzen die `LOAD` Anweisungen, die die Referenzen für den Methodenaufruf auf den Stack legen, die optimierte Variablen laden.

Ist der Rückgabewert eines optimierten Methodenaufrufs ebenfalls markiert, und wird durch eine `STORE` Anweisung in einer Variable gespeichert, so muss die Anweisung die optimierte Referenz in die entsprechende optimierte Variable speichern. Folglich wird die entsprechende Anweisung (wenn die Methode als *producing* definiert ist) durch eine `STORE` Anweisung ersetzt, die die zurückgegebene Referenz in die entsprechende optimierte Variable speichert.

5.3 Schwierigkeiten

5.3.1 Innere Archive (JARs)

Für das Lesen und Verarbeiten von Klassen eines gegebenen Programms bietet WALAs Shrike Projekt die Klasse `OfflineInstrumenter` an. Diese Klasse erwartet eine JAR Datei, und ermöglicht daraufhin eine Iteration über alle Klassen innerhalb dieses Archivs.

Befinden sich nun Klassen dieser Anwendung verpackt in einem Archiv innerhalb dieses Archivs, ist es dem `OfflineInstrumenter` nicht möglich, diese Klassen zu laden und damit auch zu verarbeiten. Daher ist das System nicht in der Lage Klassen, die nicht innerhalb des zu verarbeitenden Java-Archivs zu finden sind, zu verarbeiten.

5.3.2 Typisierung von generiertem Bytecode

Seit Java 6 können in *class* Dateien sogenannte *Stack Map Frames* verwendet werden, welche die Typisierung des Bytecodes dieser Klasse verifizieren. Diese `Stack Map Frames` dienen der Erfassung, welche lokale Variable oder Stack Position zu welchem Zeitpunkt der Ausführung, welchen Typ enthält.

Ab der Version 7 der JVM sind diese Frames nicht mehr optional sondern zwingend. Der von Shrike generierte Bytecode verpasst es allerdings diese Stack Map Frames für den neu generierten Code anzupassen. Dadurch lässt sich das

generierte Programm nur mit dem JVM Flag *-noverify* starten, welches die Typ Verifikation unterdrückt.

6 Auswertung

In diesem Kapitel soll das entwickelte System einer Auswertung unterzogen werden. Im ersten Abschnitt wird zunächst das verwendete Werkzeug *JMH* sowie die allgemeine Durchführung der Tests beschrieben werden. Im zweiten Abschnitt folgen dann die einzelnen Tests sowie deren Auswertungen.

Die Auswertung erfolgt in Form von Laufzeit Benchmarks. Dabei wird die Laufzeit von Methoden vor und nach der Optimierung gemessen und diese Ergebnisse miteinander verglichen. Außerdem werden diese Ergebnisse anhand der Ausgabe des Systems begründet.

6.1 Vorraussetzungen

6.1.1 Java Microbenchmarking Harness

Microbenchmarks sind auf der JVM schwierig zu erstellen. Das liegt vor allem an der *Just-In-Time* Kompilierung von Java Bytecode. Die JVM stellt während der Laufzeit eines Programms fest, welche Abschnitte des Bytecodes häufig ausgeführt werden, und kompiliert diese Teile dynamisch zu Maschinen Code. Diese Kompilierung kann auch wieder rückgängig gemacht werden, wenn der übersetzte Abschnitt nicht mehr so häufig oder in einem anderen Kontext verwendet wird. Zusätzlich erschwert die nicht deterministische *Garbage Collection*, während der Laufzeit, konsistente und aussagekräftige Ergebnisse.

Um diesen Problemen zu begegnen wurde das Projekt JMH verwendet. Das *Java Benchmarking Harness* (kurz JMH) ist ein Projekt innerhalb des OpenJDKs. Es dient der Unterstützung der Erstellung und Ausführung von Java Microbenchmarks. Das Framework lässt sich in den Maven Build integrieren und bietet eine annotationsbasierte Konfiguration für die in Java geschriebenen Benchmarks. Zu messende Benchmarks werden als Methoden in einer oder mehreren Java Klassen geschrieben, die eben jenen Code ausführen dessen Ausführungszeit gemessen werden soll.

6.1.2 Test Durchführungen

Alle Messungen wurden mit dem im vorherigen Abschnitt vorgestellten JMH durchgeführt. Gemessen werden soll die Ausführungszeit einer Methodenausführung in der Form vor und nach der Optimierung durch das System. Da es nicht möglich ist, dieselbe Klasse, welche die optimierte Methode bereitstellt, zweimal (die originale sowie die optimierte Variante) im Klassenpfad aufzuführen, müssen die Messungen jeweils für den originalen als auch den optimierten Methoden Aufruf separat durchgeführt werden. Diese Messungen fanden auf einem von der Universität bereit gestellten Rechner statt, der für Benchmarks zur Verfügung steht.

Für alle Tests wurden in einem Thread 10 Durchläufen durchgeführt. Ein Durchlauf setzt sich dabei aus Aufwärm- sowie einer Messphasen von jeweils 20 Iterationen zusammen. Daher existieren für jede Messung 200 Ergebnisse. Eine Iteration misst die Anzahl von Methoden Aufrufen in einer Sekunde und rechnet diesen Wert in einen Zeitwert um.

Als Test-Programme wurde zum eine eigens zum Testen geschriebene Methode verwendet, sowie Xalan, ein freier XSLT-Prozessor der Apache Software Foundation. Beide Programme wurden durch das in dieser Arbeit beschriebene System verarbeitet und für diese Auswertung verwendet.

Die Benchmarks befinden sich in dem Unterprojekt *jmh-benchmarks*, welches als Maven Eigenschaft den Pfad zur Xalan- bzw. ExampleParser-JAR Datei erwartet, und die Bibliothek in den Klassenpfad aufnimmt. So wird entsprechend des Parameters ein Benchmark entweder für die normale oder für die optimierte Variante des Java Archivs gebaut.

Für die automatische Auswertung der Ergebnisse wurde ein Skript geschrieben, das die folgenden Schritte jeweils für die normale als auch für die optimierte Variante ausführt:

1. Anstoßen der Maven Bauvorgänge für die beiden JMH-Benchmark Projekte an
2. Ausführen der resultierenden Benchmarks
3. Aufruf eines R Skript mit den erzeugten Ausgabe Dateien um die Boxplots zu erzeugen.

Alle Benchmarks wurden mit einer Oracle JVM in der Version 7 ausgeführt.

6.2 Benchmarks

In diesem Abschnitt sollen die Resultate der Messungen beschrieben und diese Ergebnisse des Systems erläutert werden.

6.2.1 Example Parser

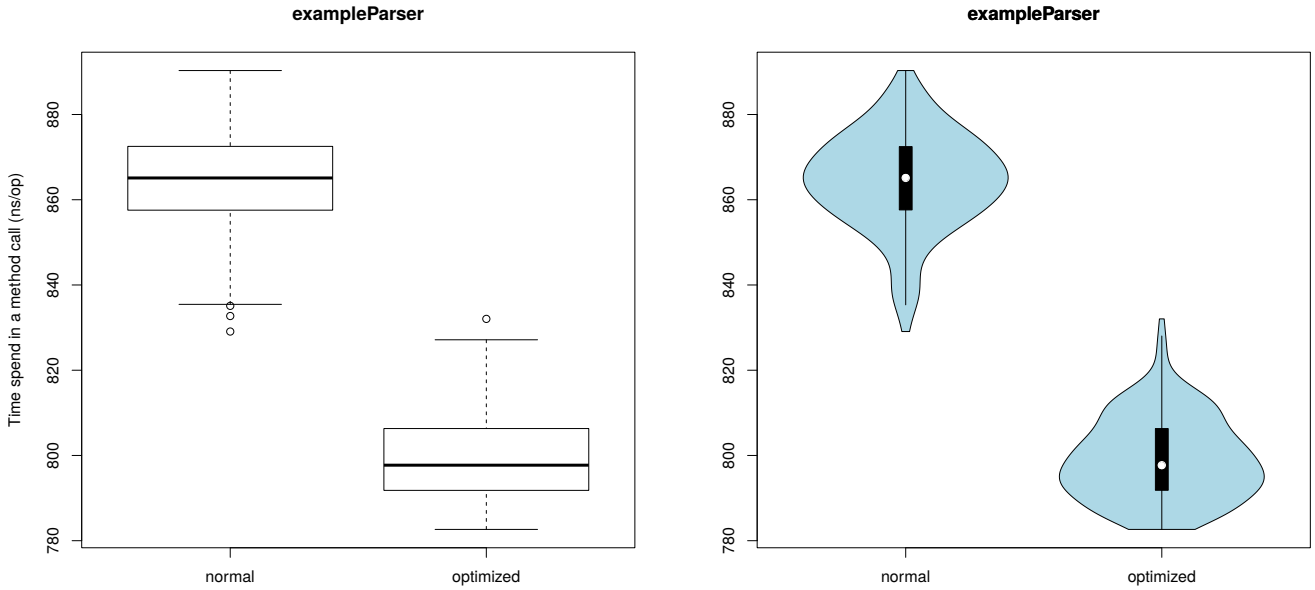
Der *Example Parser* ist eine Klasse mit einer eigens für die Auswertung geschriebenen Methode. Sie soll das Potential darstellen, dass Optimierungen mit dem in hier beschriebenen System möglich sind. Der Code stellt sich dabei wie folgt dar:

```
public String parse(String line) {
    StringBuilder result = new StringBuilder();

    for (int i = 0; i + 1 < line.length(); i+=2) {
        result.append(line.substring(i, i + 1));
    }

    return result.toString();
}
```

Abbildung 6.1: Example Parser

Wie dem Code zu entnehmen ist, wird hier die Methode `substring()` in Verbindung mit dem `StringBuilder` verwendet. Diese Methode ist daher optimal für die beiden verwendeten *TypeLabels*, welche in Kapitel 3 beschrieben sind.

Aufgerufen wurde diese Methode in einem Benchmark mit einem aus 112 Zeichen bestehenden String. Die Boxplots in Abbildung 6.2 zeigen die Dauer der Ausführung vor und nach der Optimierung durch das System.



in ns	Mittelwert	Median	$\pm$ 0,1%
normal	864,41	865,10	2,689
optimiert	799,14	797,72	2,295

Abbildung 6.2: Ergebnis des Example Parser Benchmark

Auswertung

Aufgrund der gegenseitigen Abstimmung von den beiden optimierten Typen `SubstringString` und `StringListBuilder` können die beiden Referenzen auch ohne irgendwelche Konvertierungen miteinander innerhalb der Methode koexistieren. Dies führt dazu, dass erstellte `SubstringString` Referenzen direkt dem optimierten `StringBuilder` übergeben werden können.

Dieser Benchmark soll zeigen, dass eine Optimierung, wenn auch für eine eigens dafür entwickelte Methode, möglich ist. In den folgenden Benchmarks werden realitätsnahe Methoden für den Test des Systems verwendet.

6.2.2 Xalan

Zur Auswertung des Systems wurde der freie XSLT-Prozessor *XALAN* verwendet. Dieser lag in Form eines Java Archivs vor, das aus dem Quellcode der Version 2.7.2 gebaut wurde.

Um geeignete Methoden für die Benchmarks zu finden, wurden die Anzahl der `substring(..)` Aufrufe pro `*.java` Datei gezählt und diese Kandidaten nach absteigender Anzahl sortiert. In den Kandidaten wurden diejenigen vier Methoden als Tests verwendet, welche die meisten Aufrufe von `substring` innerhalb der Methode besaßen. So fiel die Wahl auf die folgenden Methoden:

`org.apache.xalan.xsltc.runtime.BasisLibrary.checkAttribQName(String)` prüft ob der gegebene XML-Attribut Name syntaktisch valide ist.

`org.apache.xml.utils.URI.new(java.lang.String)` erzeugt ein neues URI Objekt.

`org.apache.xpath.objects.XNumber.str()` erstellt eine String Repräsentation des XNumber Objekts.

`javax.xml.xpath.XPath.compile(java.lang.String)` kompiliert den gegebenen XPath Ausdruck in ein auswertbares XPath-Objekt.

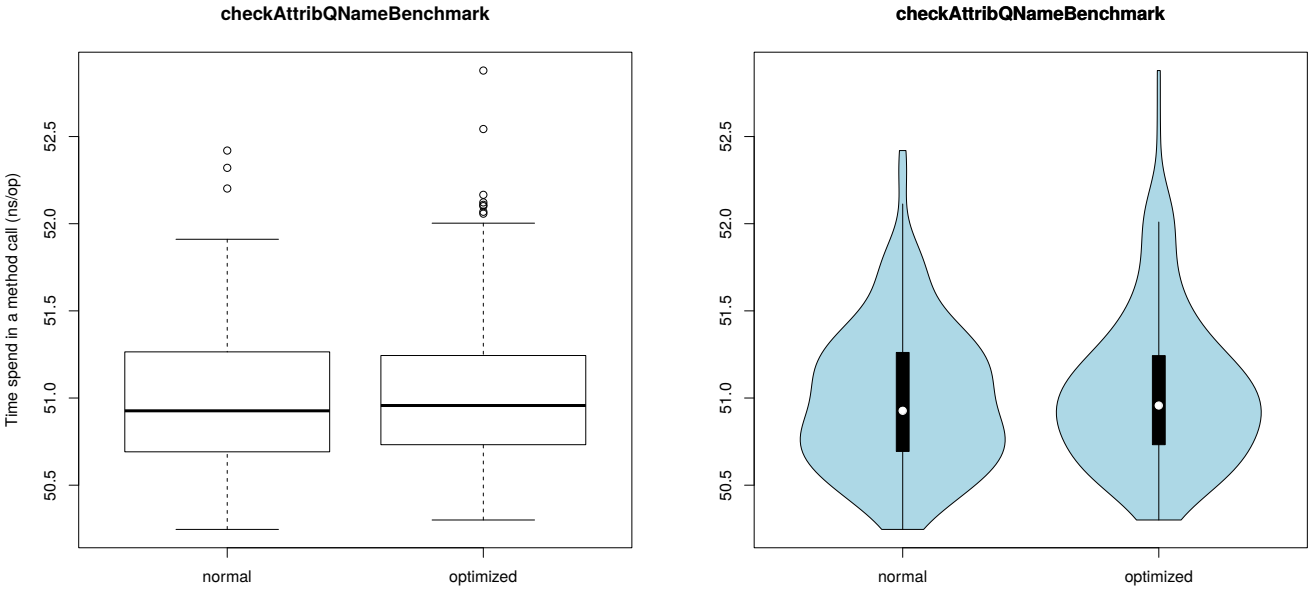
Die eigentlich identifizierte Methode ist `tokenize` des verwendeten Lexers. Da dieser aber die Sichtbarkeit package-private besitzt, wurde der kompletter Use Case, in dem der Lexer verwendet wird, für die Auswertung benutzt.

Als Eingaben dienten Werte, die der im Projekt enthaltenen Beispiel XML Dateien entnommen wurden, und daher einen möglichst realen Ausführungskontext darstellen.

checkAttribQName

Diese Funktion prüft die syntaktische Validität eines XSLT-Attribut Namens. Der Name anhand der ersten beiden ':' getrennt, was mit Hilfe der `substring` Methode geschieht.

Aufgerufen wurde diese Methode mit dem Wert 'xmlns:redirect'. Die folgende Abbildung zeigt die Ergebnisse der Messung.



in ns	Mittelwert	Median	$\pm$ 0,1%
normal	50,99	50,92	0,104
optimiert	51,03	50,96	0,108

Abbildung 6.3: Ergebnis des checkAttribQName Benchmark

Auswertung

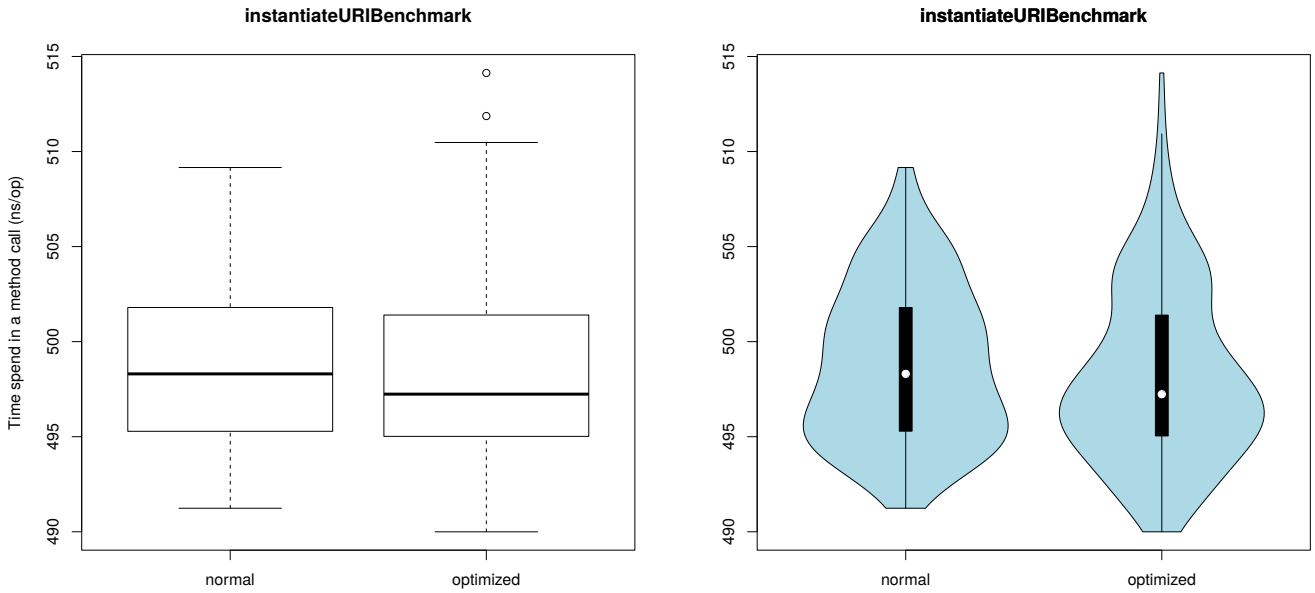
Die optimierte Referenz wird als Parameter an die Methode übergeben. Also beginnt mit diesem Parameter-Knoten die Bubble für diese Referenz wird somit zu Beginn der Methode in einen optimierten `SubstringString` konvertiert. Allerdings werden in den folgenden beiden Zeilen die Positionen der Doppelpunkte des Strings ermittelt. Die Methoden werden aber nicht von dem optimierten Typ angeboten, was dazu führt, dass an beiden dieser Stellen eine Rückkonvertierung zum originalen Typ stattfinden muss.

Dieser zusätzliche Aufwand sorgt dafür, dass die Optimierung des `substring` Aufrufs durch die Konvertierungen wieder ausgeglichen wird.

instantiateURI

Bei dieser Methode handelt es sich um den Konstruktor der Klasse `org.apache.xml.utils.URI`. Der erwartet einen String, der die zu repräsentierende URI enthält. Der Konstruktor teilt die übergebene Zeichenkette in semantische Bausteine, wie das Protokoll, die Domain oder den Pfad. Zu diesem wird die Methode `substring` wiederholt auf der übergebenen Zeichenkette angewendet.

Die Messung wurde mit dem Wert 'http://xml.apache.org/xalan-j/apidocs/javax/xml/transform/package-summary.html' durchgeführt. Die folgende Abbildung präsentiert die Ergebnisse dieses Benchmarks.



in ns	Mittelwert	Median	$\pm$ 0,1%
normal	498,79	498,30	0,965
optimiert	498,22	497,24	1.086

Abbildung 6.4: Ergebnis des instantiateURI Benchmarks

Auswertung

Bei der Optimierung dieser Methode stellen sich einige Probleme dar. Die zur Optimierung verwendete Referenz wird durch einen `trim()` Aufruf auf der übergebenen URI definiert. Diese Referenz wird innerhalb der Methode häufig benutzt, und mit jedem `substring` Aufruf auf diese Referenz wird eben diese wieder überschrieben. Zu den sonstigen aufgerufenen Methoden auf diesen Referenzen gehören:

- `startsWith(String)`
- `length()`
- `charAt(int)`

Wobei der Typ `SubstringString` `length` und `charAt` implementiert. Darüber hinaus werden diese `SubstringString` Referenzen an die folgenden privaten Methoden übergeben.

- `initializeScheme(String)`
- `initializeAuthority(String)`
- `initializePath(String)`

In diesen Methoden werden wiederum unter anderem `substring` Aufrufe auf den übergebenen String ausgeführt.

All diese anderweitigen Verwendungen der optimierten Referenz sorgen dafür, dass die Bubble an diesen Stellen endet und eine Konvertierung zum originalen Typ `java.lang.String` stattfindet. Diese Konvertierungen haben allerdings negative Auswirkungen auf die Laufzeit, wodurch die Optimierungseffekte wieder aufgehoben werden.

xNumberToString

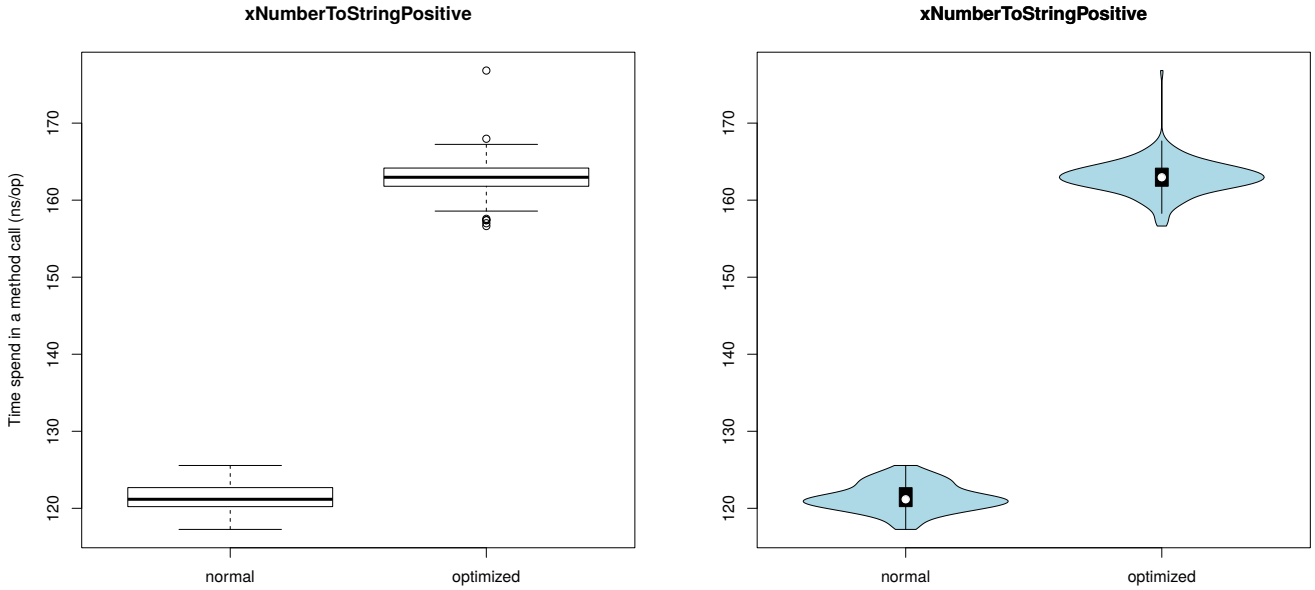
Eine `org.apache.xpath.objects.XNumber` repräsentiert eine Zahl innerhalb eines XPath Ausdrucks. Die Methode `str():java.lang.String` wandelt die intern verwaltete Dezimalzahl in eine String Repräsentation dieses Wertes um. Dabei wird das Ergebnis des Ausdrucks `Double.toString(val)` einem String zugewiesen, wobei `val` der aktuelle Wert des Objektes ist. Dieser String wird daraufhin in ein alternatives Dezimalzahlen Format umgewandelt:

- Es werden 'NaN' bzw. 'Infinity' Zeichenketten erzeugt.
- Bei Ganzzahlen wird das folgende '.0' abgeschnitten.
- Für Zahlen mit Exponenten werden diese ausgeschrieben.

Um die verschiedenen Teile der Verarbeitung innerhalb der Methode mit zu berücksichtigen, wurde der Test mit verschiedenen Eingaben durchgeführt:

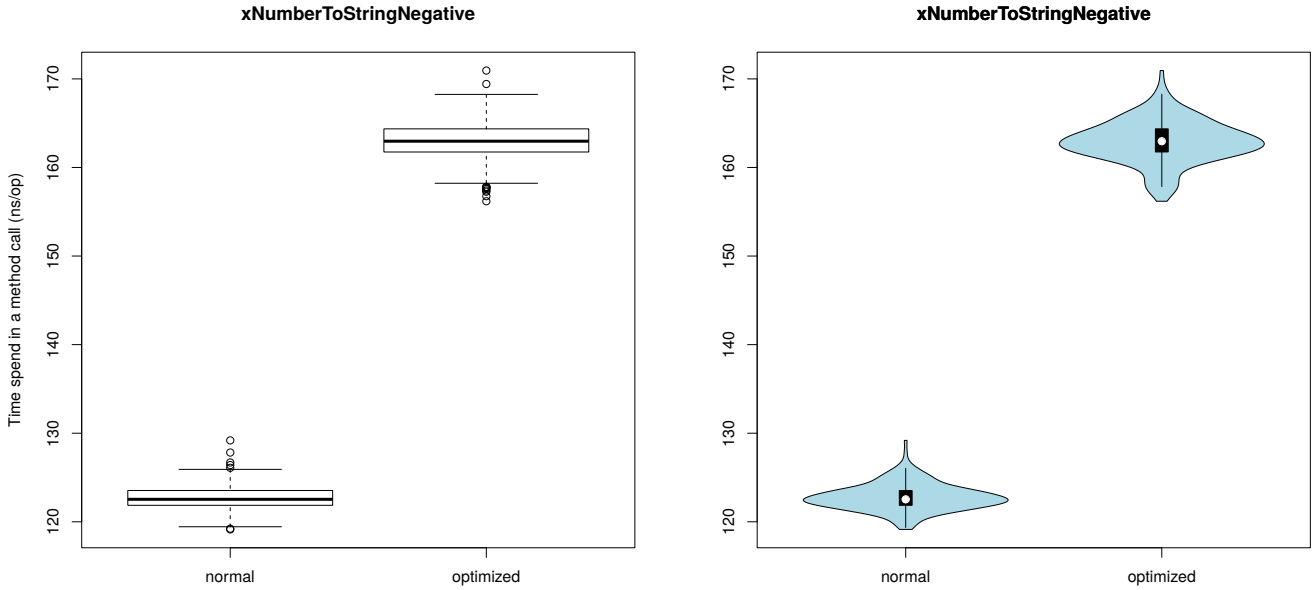
- Einer positiven Dezimalzahl (12,34)
- Einer negativen Dezimalzahl (-12,34)
- Einer positiven Ganzzahl (12)
- Einer Zahl mit negativem Exponent (0,12e-5)

Im folgenden sind die einzelnen Testergebnisse aufgeführt. Eine Auswertung der Ergebnisse folgt den Darstellungen.



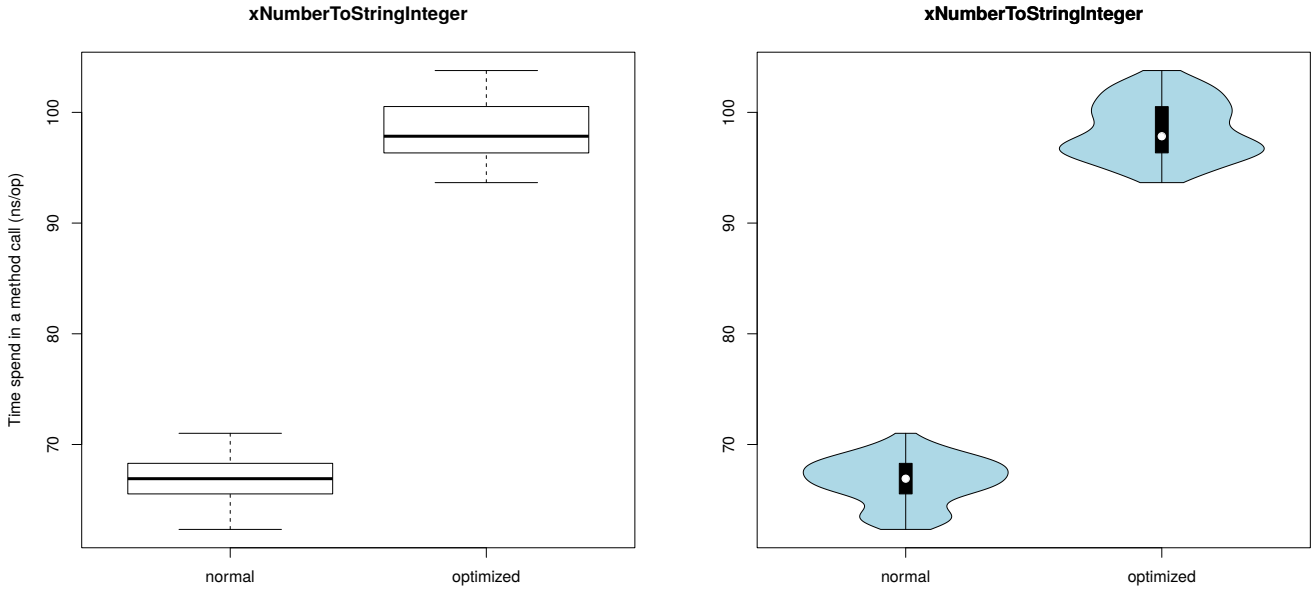
in ns	Mittelwert	Median	$\pm$ 0,1%
normal	121,38	121,17	0,425
optimiert	162,93	162,97	0,518

Abbildung 6.5: Ergebnis des xNumberToString Benchmarks (positive Dezimalzahl)



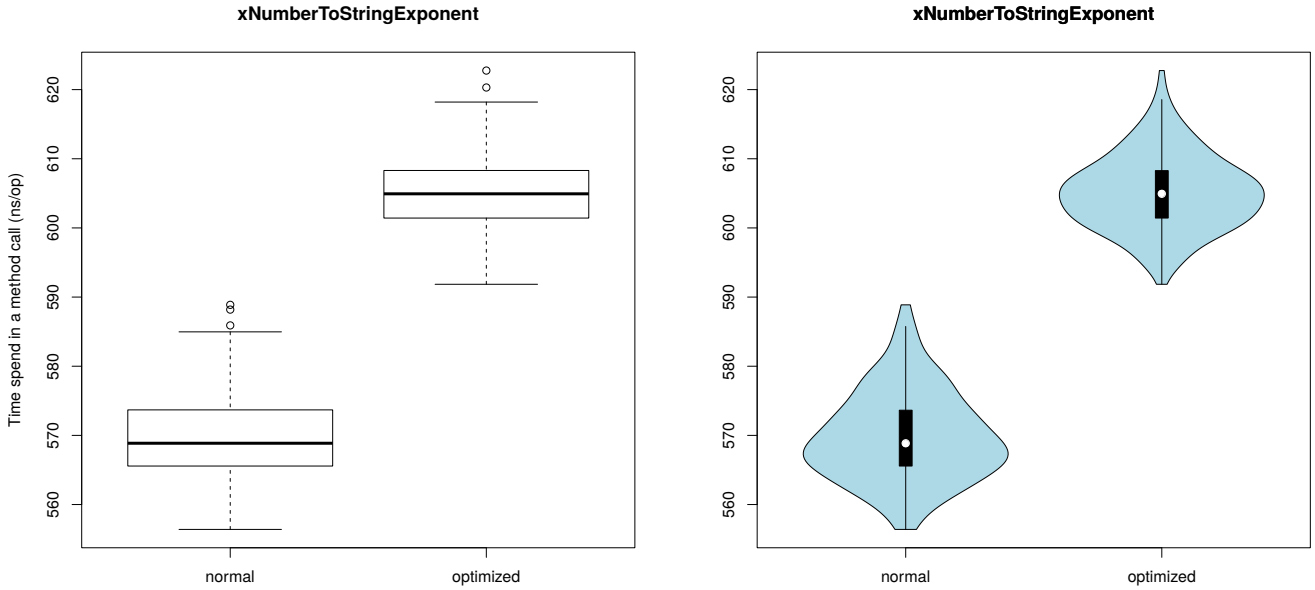
in ns	Mittelwert	Median	$\pm$ 0,1%
normal	122,78	122,54	0,359
optimiert	162,91	162,97	0,572

Abbildung 6.6: Ergebnis des xNumberToString Benchmarks (negative Dezimalzahl)



in ns	Mittelwert	Median	$\pm$ 0,1%
normal	66,69	66,91	0,488
optimiert	98,42	97,84	0,605

Abbildung 6.7: Ergebnis des xNumberToString Benchmarks (positive Ganzzahl)



in ns	Mittelwert	Median	$\pm$ 0,1%
normal	569,85	568,86	1.530
optimiert	605,16	604,93	1,291

Abbildung 6.8: Ergebnis des xNumberToString Benchmarks (Zahl mit negativem Exponent)

Auswertung

Die `xNumberToString` Methode erzeugt eine `String` Repräsentation der im Objekt verwalteten Gleitkommazahl. Der Wert ist vom Typ `double`. Der `String` wird über einen Aufruf von `Double.toString(double)` erzeugt, dem der `double` Wert der Klassen Eigenschaft übergeben wird. Die `String` Referenz wird innerhalb der Methode formatiert. Dies geschieht über Kontrollflüsse, die abhängig von der ursprünglichen Zahl sind. Die vier vorgestellten Benchmarks decken die verschiedenen Verarbeitungsschritte ab. Die auf der Referenz in jedem Fall aufgerufenen Methoden (abgesehen von `substring`) sind:

- `length()`

- `charAt(int)`
- `indexOf(char)`

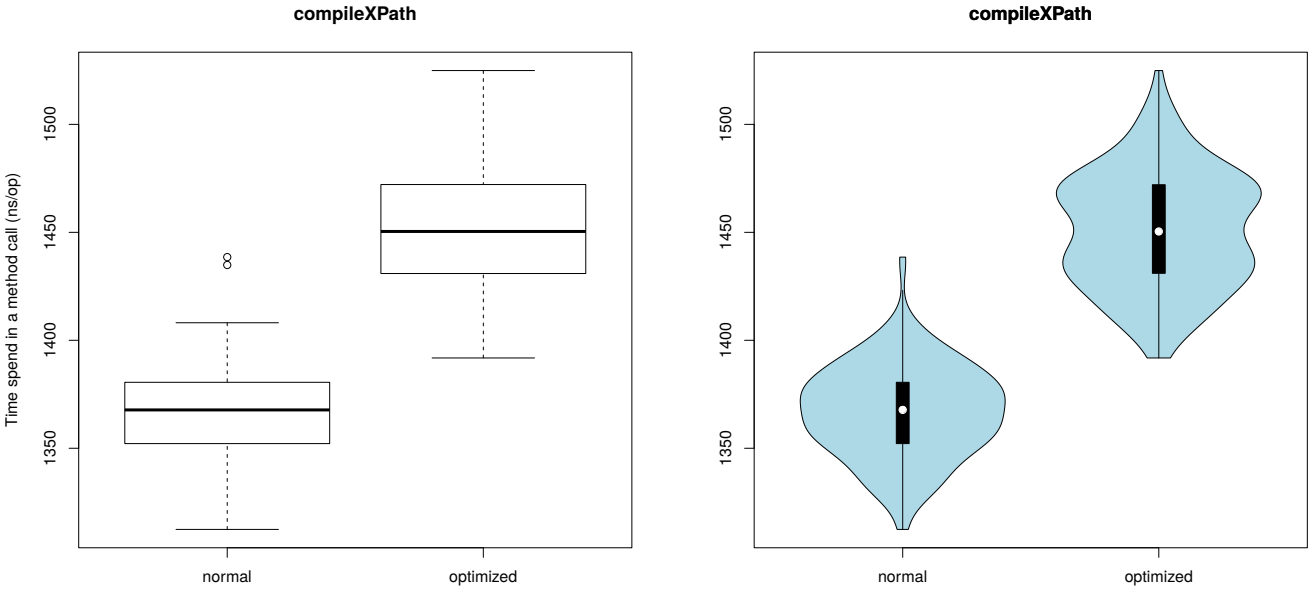
Die Methode `indexOf(char)` wird nicht von dem optimierten Typ unterstützt. Da der Aufruf allerdings für jede Zahl $\neq 0$ aufgerufen wird (es wird nach dem 'e' in Exponenten gesucht), muss jeder `SubstringString` vor dem Aufruf in einen String konvertiert werden.

Diese Konvertierungen für den Aufruf von `indexOf`, zum Beginn und zum Ende der Methode, führen zu einer beträchtlichen Anstieg der Laufzeit in der optimierten Variante dieser Methode.

compileXPath

Ein XPath Ausdruck (XML Path Language) stellt eine Abfrage gegen einen XML-Baum dar. Die Sprache ist ein Standard des W3C und stellt einen wichtigen Baustein bei der XML Transformation dar. Xalan bietet eine API, die es ermöglicht XPath Abfragen auf DOM-Bäume auszuführen. Teil dieser API ist es, einen gegebenen String in eine ausführbare Objektstruktur umzuwandeln. Zu diesem Zweck wird der gegebene `String` zunächst von einem Lexer in einzelne Token geteilt, die verwendet werden, um den abstrakten Syntax Baum des Ausdrucks zu erstellen.

Die Methode `org.apache.xpath.compiler.Lexer.tokenize(String, Vector)` führt das Zerteilen des Eingabe-Strings in einzelne Token durch. Dieser Benchmark kompiliert den XPath Ausdruck '`ex:addFunc(2, 3) + $xyz`' indem die Methode `javax.xml.xpath.XPath.compile(String)` mit dem entsprechenden `String` aufgerufen wird. Abbildung 6.9 zeigt die Ergebnisse der Messung.



in ns	Mittelwert	Median	$\pm$ 0,1%
normal	1366,91	1367,75	5,053
optimiert	1451,09	1450,40	6,346

Abbildung 6.9: Ergebnis des XPath Compile Benchmarks

Auswertung

Die `tokenize` Methode ist im Gegensatz zu den bisher betrachteten optimierten Methoden deutlich komplexer. Die einzelnen im String gefunden Token werden einer sogenannte *Token-Queue* hinzugefügt. Dies geschieht über die private Methode `addToTokenQueue`. Dabei werden bei jedem dieser Aufrufe das Ergebnis eines `substring` Aufrufs übergeben.

Eine weitere wichtige Methode während des Erzeugung der Token ist die Methode `mapNSTokens`, welcher der Eingabe String übergeben wird. Innerhalb dieser Methode werden `substring` Aufrufe auf dem übergebenen String ausgeführt.

Abgesehen von der Übergabe als Parameter werden auf den String nur unterstützte Methoden ausgeführt:

- `length()`
- `charAt(int)`

Bereits die Verwendungen der optimierten Referenzen als Übergabe Parameter zu Konvertierungen zurück zum originalen String Typ. Zusätzlich geschehen diese Konvertierungen innerhalb einer Schleife, die über alle `char`s des String läuft. Daher führen die Konvertierungen in dieser Methode zu einer erheblichen Zunahme der Laufzeit.

7 Fazit

Dieses Kapitel fasst die Ergebnisse dieser Arbeit zusammen, zieht ein Fazit über die Ergebnisse der Auswertungen und gibt Ausblicke auf zukünftige Arbeit an dem System, um die hier gezeigten Ergebnisse zu verbessern.

In dieser Arbeit wurde ein System erstellt, das anhand statischer Codeanalyse und automatischer Transformation Optimierungen anwenden kann. Es wurde die Analyse des Systems vorgestellt, in welchem Konzept des *TypeLabel*s erläutert. Das Ersetzen der originalen Datentypen durch optimierte Alternativen, sowie das Hinzufügen von Code zum Konvertieren der beiden Datentypen untereinander, wurde betrachtet und beschrieben. Schließlich wurde das System und die entwickelten String Optimierungen anhand eines einfachen Beispiels und eines komplexen Software Systems (der *Xalan* Bibliothek) getestet, indem die Laufzeit der transformierten Programme gemessen wurde.

Die Auswertung zeigt, dass die Optimierung durch automatischer Transformation von Programmen schwierig ist. Zwar lässt sich mit dem *ExampleParser* Benchmark zeigen, dass Optimierungen generell mit dem System möglich sind, doch ergeben sich Probleme bei der Anwendung in komplexeren Programmen. Das belegen die *Xalan* Benchmarks. Es zeigt sich, dass in komplexen Programmen die Kontexte, in denen Referenzen verwendet werden, die Optimierungsmöglichkeiten des Systems stark beeinflussen.

Wie in den Benchmarks *instantiateURI* und *compileXPath* zu sehen ist, ergeben sich Schwierigkeiten bei der Übergabe von optimierten Referenzen als Parameter. Da die hier erarbeitete Lösung ausschließlich intraprozeduraler Natur ist, führen derartige Benutzungen von Referenzen zwangsläufig zu zusätzlichen Konvertierungen. Der *compileXPath* Test zeigt darüber hinaus, dass zu optimierende Referenzen innerhalb des eingebetteten Methodenaufrufs ebenfalls von Optimierungen betroffen sind. Die Implementierung einer interprozeduralen Expansion der *Bubble* weist daher zusätzliches Potenzial auf, Optimierungen effizienter zu gestalten.

Die Implementierung der interprozeduralen Analyse und Transformation birgt zusätzliche Herausforderungen. So ist zunächst in der Analyse festzustellen, welche Implementierung einer Methode aufgerufen wird. Durch den

Polymorphismus, die Nutzung der *Reflections* API und dynamisches Klassenladen zur Laufzeit, ist dies nicht über statische Code Analyse alleine zu lösen, sondern erfordert eine Analyse zur Laufzeit des Programms. Mittels dieses Ansatzes wäre es möglich die Ziele von abstrakten Methodenaufrufen zu identifizieren und diese für die Analyse zu verwenden.

Ein weiteres Problem ergibt sich bei der Transformation in einem interprozeduralen Kontext. Bei Methoden, deren Parameter in optimierter Form übergeben werden, muss die Signatur angepasst werden, um kompatibel mit dem optimierten Typ zu sein. Handelt es sich bei der Methode allerdings um einen Teil der öffentlichen API, so darf diese Signatur nicht ersetzt werden. Damit würde ein Klienten Programm, das diese Methode verwendet und selber nicht optimiert wurde, zu Laufzeitfehler führen.

Das in dieser Arbeit erstellte System bietet durch die *TypeLabel* Schnittstelle, die Möglichkeit verschiedenste Optimierungen von Typen zu verarbeiten. Daher ist es nicht auf die Optimierung von Strings beschränkt. Da diese Arbeit ausschließlich auf die von Optimierungen für die String API eingeht, wurden auch keine anderen Optionen in Erwägung gezogen. Dabei schließt das System die Anwendung alternativer Optimierungen nicht aus, sondern ist auf alle Typen des Java Typ Systems anwendbar.

Als Kandidaten wäre die Dezimalzahl Repräsentation `java.math.BigDecimal` zu betrachten, die zwar exakte Darstellung von Nachkommastellen, aber nicht die Performanz eines `float` oder `double` Wertes, bietet. Es wäre eine Repräsentation des `BigDecimal` Wertes denkbar, die für Darstellung einer Ganzzahl optimiert ist. Dabei ergibt sich die Schwierigkeit, dass mit Hilfe von statischer Code Analyse nicht identifizierbar ist, ob eine Variable eine Ganzzahl oder eine Dezimalzahl darstellt. Für eine solche Analyse müsste ebenfalls die dynamische Code Analyse verwendet werden, die feststellt, wie eine Referenz während der Laufzeit verwendet wird und welche Werte sie annimmt.

Literaturverzeichnis

[1] CoCo: Sound and Adaptive Replacement of Java Collections, Guoqing Xu, ECOOP 2013

[2] Chameleon: Adaptive Selection of Collections, Ohad Schacham, Martin Vechev, Eran Yahav, PLDI 2009

[3] Brainy: Effective Selection of Data Structures, Changhee Jung, S. Rus, B. Railing, N. Clark, Santosh Pande, PLDI 2011

[4] `http://wala.sourceforge.net/wiki/index.php/Main_Page` (Stand 10.11.2014)