Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 189

Bestimmung der Ausführungszeit von Java-Anwendungen zur Laufzeit

Oliver Kabierschke

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. net. Kurt Rothermel
Betreuer/in:	Dipl.-Inf. Florian Berg
Beginn am:	19. November 2014
Beendet am:	21. Mai 2015
CR-Nummer:	D.2.8, D.4.8

Kurzfassung

Das Auslagern von Programmcode stellt eine Möglichkeit dar, die Ausdauer und Leistungsfähigkeit akkubetriebener Mobilgeräte zu verbessern. Um feststellen zu können, ob sich das Auslagern lohnt, ist es unter anderem notwendig, die Ausführungszeit von Programmteilen zu bestimmen. In dieser Arbeit wird hierzu eine Verfahrensweise vorgestellt, die ohne Zugriff auf den Quellcode eines laufenden Java-Programms auskommt. Dabei wird durch statische Analyse von Java-Methoden die Häufigkeit der Ausführung ihrer Abschnitte ermittelt und mit Messergebnissen der einzelnen Anweisungen auf die Ausführungsdauer der gesamten Methode geschlossen. Bei der Messung solcher Anweisungen, mit denen die Java Virtual Machine instruiert wird, treten Probleme auf, zu denen diese Arbeit Lösungsansätze und eine mögliche Implementierungsweise liefert. Es wird weiterhin gezeigt, wie durch dynamische Analyse die so gewonnenen Ergebnisse zur Laufzeit verbessert werden können. Aus dieser Arbeit resultiert eine Entscheidungsgrundlage für die Offloading-Komponente, mit der diese fundiert entscheiden kann, ein Programmteil lokal auszuführen oder zu einem entfernten Server zu übertragen, um die Berechnung dort durchführen zu lassen und so Energie zu sparen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Code-Auszüge

1. Einleitung

Ein Großteil der Kommunikation läuft heutzutage über das Smartphone, welches aus unserem Alltag nicht mehr weg zu denken ist. Wir haben es zu jeder Zeit bei uns, da es vor allem die Möglichkeit eröffnet, unterwegs Kontakt zu Kollegen, Freunden und Familie zu halten. Daneben dient es ebenfalls als Informationsquelle und Unterhaltungsmedium. Moderne Geräte sind mit großem Speicher, hochauflösendem Display und leistungsfähigem Prozessor ausgestattet. Dabei werden Software und mobile Datenverbindung immer zuverlässiger, sodass das Mobiltelefon jederzeit einsatzbereit ist – solange der am meisten limitierende Faktor nicht ausfällt: Die Energieversorgung. Im besten Fall versorgt ein durchschnittlicher Akku ein Smartphone einen Tag lang mit elektrischem Strom; bei intensiverer Nutzung erfordert die Energiequelle jedoch bereits nach wenigen Stunden das Aufsuchen einer Steckdose. Zu den größten Stromverbrauchern zählt neben dem Bildschirm auch der Prozessor. Besonders aufwändige Anwendungen, wie grafikintensive Spiele, fordern von den Recheneinheiten Höchstleistung ab, die den Energieverbrauch in die Höhe treibt.

Gegenstand aktueller Forschung ist daher die Auslagerung von Programmcode auf entfernte Server. Das kann sowohl helfen, die Laufzeit mobiler Geräte zu verlängern [SSX+12], als auch die Ausführung komplexer Anwendungen zu beschleunigen [BDR14], da Prozessoren in Servern in der Cloud natürlich leistungsfähiger sind, als mobile Exemplare. Die Möglichkeiten bei der Steigerung der Leistung mobiler Anwendungen gehen bis hin zu einer Verdopplung der Bildraten grafisch aufwendiger Spiele und dem Erstellen neuer Anwendungen, die ohne die zusätzliche Rechenleistung auf Mobilgeräten erst gar nicht realisierbar wären, wie beispielsweise eine Echtzeit-Übersetzung von gesprochener Sprache [CBC+10].

Die Ausführungsschicht eines mobilen Endgerätes muss nun entscheiden, ob sich das Übertragen eines Programmfragmentes in die Cloud lohnt, oder ob eine lokale Berechnung günstiger wäre. Dabei spielen Faktoren, wie die gegenwärtige Netzwerkanbindung (Qualität der mobilen Datenverbindung oder des WLAN) und die Auslastung der Cloud-Server eine Rolle. Genauso bedeutend für eine Entscheidungsfindung ist der Vergleich zwischen dem zum Übertragen notwendigen Energieaufwand und demjenigen zum lokalen Ausführen einer Funktionalität. Würde das Übertragen und Warten auf die entfernte Berechnung mehr Energie verbrauchen, als das Ausführen einer Funktion auf dem lokalen Prozessor, wird sich eine Offloading-Komponente kaum zum Auslagern entscheiden. Die Schwierigkeit liegt nun im Finden dieser verglichenen Größen.
Während sich der Energieaufwand zum Übertragen des Programmfragments aus der Größe der zu übertragenden Daten und der Netzwerkanbindung errechnen lässt [FN01], ist beim Berechnen des Aufwands zum lokalen Ausführen eines Programms vor allem die Ausführungszeit auf dem Prozessor des Gerätes von Bedeutung. Eine einfache Methode, an diese Größe zu kommen, ist das Messen der Dauer einer Ausführung des auszulagernden Programmteils. So kann die Offloading-Software beim wiederholten Aufruf einer Funktionalität auf die gespeicherten Messdaten zurückgreifen, um den voraussichtlichen Energiebedarf statistisch zu ermitteln. Diese Herangehensweise scheitert jedoch

sowohl dann, wenn keine gespeicherten Daten vorhanden sind und eine Funktion das erste Mal ausgeführt wird, als auch bei stark schwankenden Messergebnissen.
In beiden Fällen ergibt es Sinn, die Programmfragmente vor ihrer Ausführung zu analysieren, um so einen ersten Richtwert ihrer Dauer zu erhalten. Genauso können bei dieser Analyse von der Methode verwendete variable Daten identifiziert werden, die erst zur Laufzeit unmittelbar vor ihrer Ausführung feststehen und zu diesem Zeitpunkt möglicherweise weitere Hinweise auf die Ausführungsdauer geben.

Die Aufgabenstellung dieser Arbeit soll daher die Analyse von kompilierten Programmen sein, um vor der ersten Ausführung eine Vorstellung von ihrer voraussichtlichen Ausführungszeit zu bekommen.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen stellt für das Verständnis dieser Arbeit notwendige Methoden und Techniken vor.

Kapitel 3 – Bestimmung der Ausführungszeit von Java-Bytecodes beleuchtet Probleme, die beim Messen der Ausführungszeit einzelner Java-Bytecode-Instruktionen auftreten und gibt Lösungsansätze.

Kapitel 4 – Analyse von Java-Bytecodes demonstriert Methoden zur Analyse von Java-Programmen, die in Form von Bytecode-Instruktionen vorliegen.

Kapitel 5 – Evaluation prüft die gezeigten Ansätze auf deren Umsetzbarkeit und analysiert die Effizienz der gegebenen Implementierungsmöglichkeiten.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2. Grundlagen

Dieses Kapitel soll zum Verständnis des Themas dieser Arbeit vorausgesetztes Wissen vermitteln. So werden eine Einführung in die Besonderheiten von Java-Programmen gegeben und Problemstellungen beim Code-Offloading vorgestellt. Weiterer Bestandteil der Ausführungen in diesem Kapitel ist Software, welche wesentlich zum Erreichen des Ziels der Arbeit beigetragen hat.

2.1. Java

Nach dem Schreiben von Quellcode in herkömmlichen Programmiersprachen, wie C und C++, werden die Quelldateien von einem Compiler in ein maschinenlesbares Format übersetzt. Der dabei entstandene Maschinencode ist auf das Gerät zugeschnitten, auf dem er ausgeführt werden soll. Prozessoren haben unterschiedliche Befehlssätze, was die Existenz von speziellen Compilern für jede Prozessor-Architektur erforderlich macht. Ein C-Programm, das für einen PC mit x86-Prozessor kompiliert wurde, kann folglich nicht auf einem Smartphone mit ARM-Prozessor ausgeführt werden. Hierzu wäre ein erneutes Übersetzen des Quellcodes mit einem dafür konzipierter Compiler nötig.

Die Sprache Java funktioniert grundlegend anders. Hier wird aus dem Quellcode kein Maschinencode generiert, sondern plattformunabhängiger Java-Bytecode. Zur Ausführung dieses Zwischencodes existieren auf die Zielplattform zugeschnittene Ausführungsschichten, die Java Virtual Machines (JVM) [LY99]. Eine JVM übersetzt den Java-Bytecode in den Maschinencode der gegebenen Plattform und bringt diesen zur Ausführung. So wird erreicht, dass ein Java-Programm ohne erneutes Kompilieren auf jeder beliebigen Architektur ausgeführt werden kann. Die einzige Voraussetzung dafür ist die Existenz einer JVM für die spezifische Plattform.

Als Transport-Format für den Java-Bytecode dient das *Java Archiv*, welches wegen seiner Dateiendung auch JAR-Archiv genannt wird. Der Java-Compiler erzeugt ein solches aus den vom Programmierer geschriebenen Quellcode-Dateien; die JVM liest das Archiv schließlich ein, übersetzt es in Maschinencode und führt diesen aus. Beim Java Archiv handelt es sich um einen ZIP-Kontainer, der in einer festgelegten Ordnerstruktur *class*-Dateien beinhaltet, in welchen der Java-Bytecode der Klassen des Programmes gespeichert ist.
Als objektorientierte Sprache [RBP+91] ist der Quellcode von Java in Klassen und Methoden organisiert. Für jede Klasse wird eine solche *class*-Datei angelegt, die mit dem Namen der Klasse und der Endung „.class“ benannt ist. In jedem Java-Programm gibt es eine Hauptklasse, in welcher der Startpunkt der Anwendung in Form der Methode *main* liegt. Damit die JVM die Hauptklasse findet und die Ausführung dort starten kann, wird ihr Name in der Manifest-Datei des JAR-Archivs „META-INF/MANIFEST.MF“ gespeichert. Klassen sind weiterhin in Paketen (engl. packages) organisiert. Pakete können Klassen und weitere Pakete enthalten. Ihre Struktur wird in die Ordnerstruktur des JAR-Archivs umgesetzt.

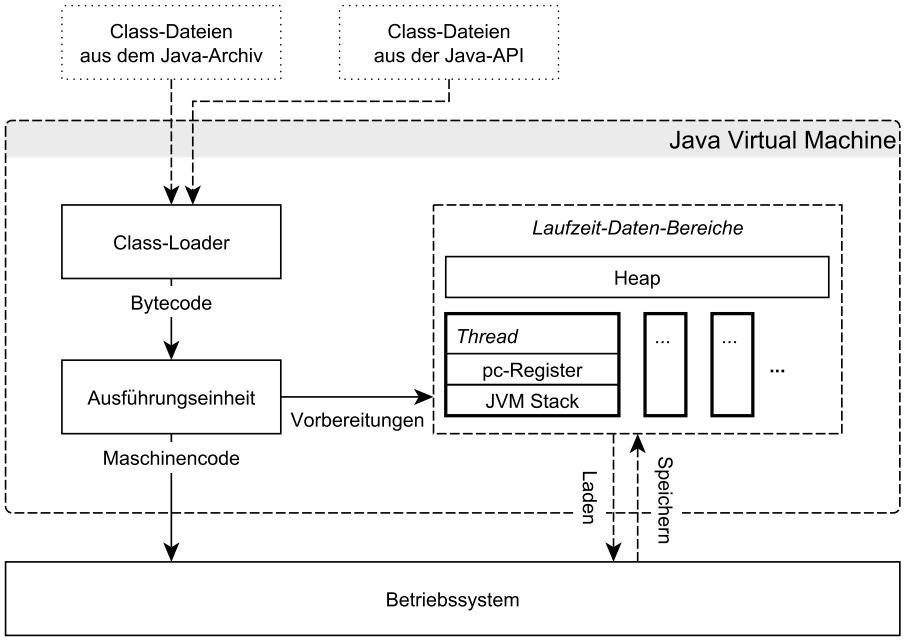


Abbildung 2.1.: Die Java Virtual Machine in der Übersicht

2.1.1. Die Java Virtual Machine

Zum Ausführen und kompilieren von in JAR-Archiven abgelegten Java-Programmen ist die JVM in Komponenten unterteilt, die unterschiedliche Teilaufgaben übernehmen. Einen Überblick über die einzelnen Teile und deren Zusammenwirken gibt Abbildung 2.1. Im Folgenden wird die Funktionsweise einer Java Virtual Machine grundlegend erklärt.

Start einer Java-Anwendung

Wird eine JVM gestartet, etwa durch einen Befehl wie `java -jar App.jar`, erzeugt sie Class-Loader, welche die *class*-Dateien aus dem auszuführenden JAR-Archiv in eine interne Darstellung von Java-Klassen überführen. Der Bytecode der Methoden einer Klasse wird dabei nicht verändert und ist daher noch nicht ausführbar. Erst die Ausführungseinheit setzt ihn in Maschinencode um, bereitet den Methodenaufruf vor und initiiert die Ausführung auf dem Prozessor. Dieses Verhalten ist das eines Interpreters, jedoch ist eine JVM vielfach schneller als eine klassische Interpreter-Sprache wie Python oder PHP, da sie vor allem die Java-Bytecodes nicht mehr auf syntaktische Fehler überprüfen muss. Solche Prüfungen sind Aufgabe des Java-Compilers, der keinen Quellcode zu Java-Bytecodes umsetzt, der nicht syntaktisch fehlerfrei ist.

Für den Zugriff auf Systemfunktionen, wie das Lesen und Schreiben von Dateien auf der Festplatte oder das Auslesen der Uhrzeit, bietet jede JVM eine wohldefinierte Menge an Methoden an. Diese

Menge heißt Java-API und wird genau wie die auszuführende Anwendung durch einen Class-Loader geladen.

Außerdem werden beim Start Laufzeit-Daten-Bereiche angelegt, in welchen für die Ausführung des Programms wichtige Daten abgelegt werden. Dabei gibt es den Heap (vgl. Abbildung 2.1), der von allen Threads (s. u.) gemeinsam verwendet wird und auf welchem vom Java-Programm erzeugte Objekte und Arrays abgespeichert werden. Außerdem enthält der Heap den Methodenbereich (engl. method area), in welchem für jede Klasse dessen Laufzeit-Konstanten-Pool und der Code ihrer Methoden liegen. Im Laufzeit-Konstanten-Pool finden sich feste Werte, die bei der Ausführung von Methoden benötigt werden und nicht im Bytecode abgelegt sind, wie Zeichenketten und Referenzen zu Klassen. Diese Konstanten können von den Instruktionen einer Methode geladen und verarbeitet werden.

Threads

Die Ausführung eines Java-Programms ist in einen oder mehrere Threads gegliedert, die je nach Prozessorarchitektur der zugrunde liegenden Plattform gleichzeitig ablaufen können. Läuft die JVM beispielsweise auf einem System mit einem Dual-Core-Prozessor, können zwei Threads zur selben Zeit ablaufen. Das Betriebssystem ist darüber hinaus jedoch in der Lage, mehr als diese zwei Threads quasi-parallel auszuführen, indem sie zu gewissen Zeitpunkten die Ausführung eines Threads pausiert und einen anderen an dessen Stelle startet. Jeder Thread hat zu jedem Zeitpunkt genau eine aktive Methode. Die gegenwärtige Methode wechselt, sobald diese eine weitere aufruft oder mit ihrem Ablauf am Ende ist. Im letzteren Fall wird die Ausführung an der Stelle fortgesetzt, an welcher die gegenwärtige Methode aufgerufen wurde.

Für jeden Thread legt die Ausführungseinheit bei dessen Erzeugung weitere Laufzeit-Daten-Bereiche an. Dort sind unter anderem der Methodenzähler (pc-Register), der den Fortschritt in der aktuell ausgeführten Methode anzeigt, sowie der JVM Stack untergebracht, auf welchem Daten zu Methodenaufrufen gespeichert werden.

Stacks (zu deutsch: Stapel) sind Datenstrukturen, bei denen das Element, welches als erstes gespeichert wird, als letztes aus ihr entfernt wird. Es handelt sich also um eine LIFO-Datenstruktur (Last In - First Out, englisch für zuletzt hinein - zuerst heraus). Wird vom „oberen Ende“ eines Stacks gesprochen, ist das Element gemeint, welches als letztes „auf den Stack gelegt“ wurde. Analog dazu bezeichnet das „untere Ende“ das Element, welches am längsten in der Datenstruktur verweilt.
Ein Heap (zu deutsch: Halde, Haufen) ist im Gegensatz dazu ein ungeordneter Speicher mit wahlfreiem Zugriff. Objekte darin werden durch eine Adresse angesprochen.

Auf dem *JVM Stack* legt die Ausführungseinheit beim Aufruf einer Methode einen *Frame* für diese an. Ruft die laufende Methode eine weitere Methode auf, liegt auf dem JVM Stack ein weiteres Frame. Beim Ende einer Methodenausführung wird das Frame der beendeten Methode entfernt. In einem solchen Frame sind Daten gespeichert, die während oder nach der Ausführung der dazugehörigen Methode erforderlich sind. So findet man dort die von ihr verwendeten lokalen Variablen, sowie den *Operand-Stack*, welcher auch als Zwischenspeicher für Arithmetische Operationen dient (siehe 2.1.2 – Java-Bytecodes).

Variablen und Felder

Eine Methode kann bei ihrer Ausführung auf Variablen und Felder zurückgreifen, um Daten zu speichern und später wiederzuverwenden. Für diese Variablen stehen die folgenden Datentypen zur Verfügung:

byte: Eine 8-bit Ganzzahl, dargestellt als Zweierkomplement.

short: Eine 16-bit Ganzzahl, dargestellt als Zweierkomplement.

integer: Eine 32-bit Ganzzahl, dargestellt als Zweierkomplement.

long: Eine 64-bit Ganzzahl, dargestellt als Zweierkomplement.

char: Eine 16-bit Ganzzahl ohne Vorzeichen, die ein Zeichen repräsentiert.

float: Eine 32-bit Gleitkommazahl.

double: Eine 64-bit Gleitkommazahl.

Außerdem findet der Typ *reference* Verwendung, dessen Werte Referenzen auf Objekte sind.

Methodenkompilierung

Damit eine Methode auf dem Prozessor ausgeführt werden kann, muss sie in dessen Befehlssatz überführt werden. Entscheidend für die Leistungsfähigkeit einer JVM ist der Zeitpunkt, zu dem sie eine Methode kompiliert. So hat sie die Möglichkeit, für alle Methoden des Java-Programms vor der Ausführung der ersten Methode, deren Maschinencode-Repräsentation zu generieren. Häufiger findet man in den gegenwärtigen Implementierungen von Virtual Machines den *Just-In-Time-Compiler (JIT-Compiler)*. Hier wird eine Methode erst kompiliert, sobald sie das erste Mal aufgerufen wird. Das beschleunigt in jedem Fall den Start der Anwendung.

2.1.2. Java-Bytecodes

Java-Bytecodes definieren den Programmablauf einer Methode in einem Java-Programm, wie ihn der Programmierer vorgesehen hat. Er hat Ähnlichkeiten mit Assembler-Code (Maschinencode), was ein Übersetzen in den Befehlssatz des Prozessors durch die JVM deutlich einfacher gestaltet, als wenn die JVM den ursprünglichen Quellcode in Maschinencode kompilieren müsste. Jeder Bytecode besteht aus 8 Bit und kann somit als eine Zahl zwischen 0 und 255 dargestellt werden. Die Zuordnung dieser Zahlen ist eindeutig, es gibt also zu jeder Bytecode-Instruktion genau einen 8-Bit-Wert als dessen Bezeichner. In den *class*-Dateien wird der Ablauf von Methoden in Form dieser Bezeichner abgelegt. Als Aneinanderreihung von Zahlen ist dieses Format sehr speichereffizient.

Die Auflistung und Erklärung der einzelnen Instruktionen nimmt viel Platz ein und wurde aus diesem Grund in den Anhang unter Punkt A.1 verlagert. Im Folgenden soll eine für das Verständnis dieser Arbeit wichtige Zusammenfassung mit den wichtigsten Aspekten von Java-Bytecodes gegeben werden.

Arithmetische Operationen

Die Befehle, welche eine Java Virtual Machine ausführen kann, haben mit wenigen Ausnahmen Auswirkungen auf den Operand-Stack der aktuellen Methode. Es gibt produzierende Anweisungen, welche einen oder mehrere Werte auf den Stack schreiben, konsumierende Anweisungen, welche von dort einen oder mehrere Werte entfernen, und solche, die sowohl Werte entfernen, als auch schreiben.

Viele Instruktionen erwarten zusätzliche Informationen als Parameter. Ein solches Beispiel ist `iload`, die einen Wert vom Typ *integer* aus den lokalen Variablen auf den Operand-Stack lädt. Der Parameter, welcher als weiteres Byte im Bytecode direkt hinter der Instruktion folgt, gibt an, aus welcher lokalen Variable gelesen werden soll.
Genauso erwartet `bipush` als darauffolgendes Byte den Wert, welchen die Instruktion als *integer* auf den Operand-Stack schreiben soll.
Von Instruktionen zum Laden und Speichern von lokalen Variablen gibt es zum Sparen von Speicherplatz und zum Steigern der Effizienz beim Umsetzen des Codes parameterlose Formen. So kann eine der ersten vier lokalen Variablen durch einen der Bytecodes `istore_0`, `istore_1`, `istore_2` und `istore_3` beschrieben werden. Mit der Instruktion `istore` wären dazu zwei Bytes notwendig.
Weiterhin existieren Instruktionen, die mehrere Werte vom Operand-Stack auf einmal konsumieren. Die Anweisung zum Addieren zweier *integer*-Werte `iadd` ersetzt z. B. die beiden obersten Werte des Operand-Stacks durch die aus ihnen berechnete Summe.

Ein simples Java-Programm in Bytecode-Form, das zur ersten lokalen Variable den Wert 12 addiert und das Ergebnis in die fünfte lokale Variable ablegt, wird in Code-Auszug 2.1 dargestellt.
Der Wert in der ersten Spalte ist derjenige, der tatsächlich in einer *class*-Datei vorzufinden wäre. In der zweiten Spalte wird die Bedeutung des Wertes erläutert. Hinter Instruktionen steht so deren Name. Ist das Byte ein Argument einer vorherigen Instruktion, wird dies durch `<ARG: x>` verdeutlicht, wobei `x` der übergebene Wert ist.
In der dritten Spalte kann die Zustandsänderung des Operand-Stacks verfolgt werden, die durch die jeweilige Instruktion bewirkt wird. Dabei wird angenommen, dass vor der Ausführung des Programms der Wert 4 in der ersten lokalen Variable gespeichert ist.

Code-Auszug 2.1 Ein einfaches Java-Programm in Form von Bytecodes

```
1  0x1a  iload_0     []->[4]        // Lade die erste lokale Variable.
2  0x10  bipush      [4]->[4,12]    // Lade den konstanten Wert...
3  0x0c  <ARG: 12>               // ...12 auf den Operand-Stack.
4  0x60  iadd        [4,12]->[16]   // Addiere die beiden Zahlen auf dem Stack.
5  0x36  istore      [16]->[]       // Speichere das Ergebnis...
6  0x05  <ARG: 5>                // ...in die Variable 5.
```

Flusskontrolle

Eine weitere, wichtige Gruppe an Bytecode-Instruktionen sind verzweigende Anweisungen. Bei ihrer Ausführung wird darüber entschieden, ob regulär mit der nächsten Anweisung fortgefahren, oder ob der Programmablauf an einer anderen Stelle fortgesetzt wird. Während die Instruktion `goto`

(S. 50) eine unbedingte Verzweigung erzeugt und die Ausführung in jedem Fall an der in ihrem Parameter definierten Stelle fortgesetzt wird, vergleichen `if_icmp<bdg>` und `if<bdg>` (S. 51) Werte vom Operand-Stack, sodass bei positivem Vergleich die Verzweigung durchgeführt wird und sonst die im Programmablauf nachfolgende Bytecode-Instruktion zur Ausführung kommt.

Abschließend beenden Instruktionen, wie `<t>return` (S. 50) die aktuelle Methode, woraufhin die Ausführung an der Stelle fortgesetzt wird, an welcher diese durch eine Instruktion wie `invokestatic` (S. 53) aufgerufen wurde.

2.2. Code-Offloading

Um auf mobilen Geräten, wie Smartphones oder Tabletcomputern, Energie zu sparen, können Teile von Anwendungen auf entfernte Server übertragen und dort zur Ausführung gebracht werden. Dadurch verkürzt sich im Optimalfall die Rechenzeit auf dem mobilen Gerät und der Energieverbrauch wird gesenkt. Da die Prozessoren in Servern um ein Vielfaches schneller sind, als die in mobilen Geräten, ist dabei sogar eine deutliche Beschleunigung der Ausführung von Anwendungen erzielbar. Ein Ansatz zur Realisierung von Code-Offloading besteht in der Integration einer Offloading-Komponente in die Anwendungssoftware, d.h. die Apps, welche auf mobilen Geräten ausgeführt werden. Diese Komponenten würden mit einer kompatiblen Anwendung auf einem Server in der Cloud kommunizieren, um dort Rechenarbeit ausführen zu lassen. Ebenso ist es möglich, Anmerkungen in die App zu integrieren, die von einer Offloading-Software auf dem mobilen Gerät gelesen und für die Entscheidung zum Auslagern genutzt werden. In beiden Fällen kann und muss der Programmierer einer App entscheiden, für welche Teile seiner Anwendung es besonders lohnend ist, diese auf einem schnellen Server auszuführen.
Wollte man auf eine dieser Arten Code-Offloading betreiben, müssten bereits existierende Anwendungen so angepasst werden, dass sie den neuen Standard unterstützen. Offensichtlich ist es nicht praktikabel, von einer Anwendung für Mobilgeräte zu erwarten, dass ihr Programmierer Vorkehrungen zum Auslagern dieser getroffen hat. Gesucht ist folglich ein Weg, bei einem beliebigen Programm Teile zu erkennen, die ausgelagert werden können, sowie das Ergebnis der entfernten Ausführung in die Anwendung zu installieren, sodass die Software mit dem regulären Programmablauf fortfahren kann.

2.2.1. Die Offloading-Komponente

Wir haben im vorangegangenen Abschnitt über die Besonderheiten von Java gelernt: Der Quellcode wird zu Zwischencode kompiliert, welcher erst auf der Ausführungsschicht auf dem Zielsystem zu Maschinencode umgesetzt wird. Die Eigenheit, dass es in Java eine Virtual Machine gibt, die die Ausführung von Java-Programmen koordiniert, wird nun für das Auslagern von Programmcode ausgenutzt. Die VM kann so angepasst werden, dass sie anstatt Methoden nur auszuführen, diese analysiert und entscheidet, ob sich eine Auslagerung auf einen entfernten Server lohnt.

Das Auslagern von Programmteilen wird somit praktischerweise von einer Komponente realisiert, die innerhalb der JVM untergebracht ist. Dort wird sie vom Ausführungsteil der Virtual Machine über Methodenaufrufe informiert und kann daraufhin Informationen zu der Methode sammeln, die

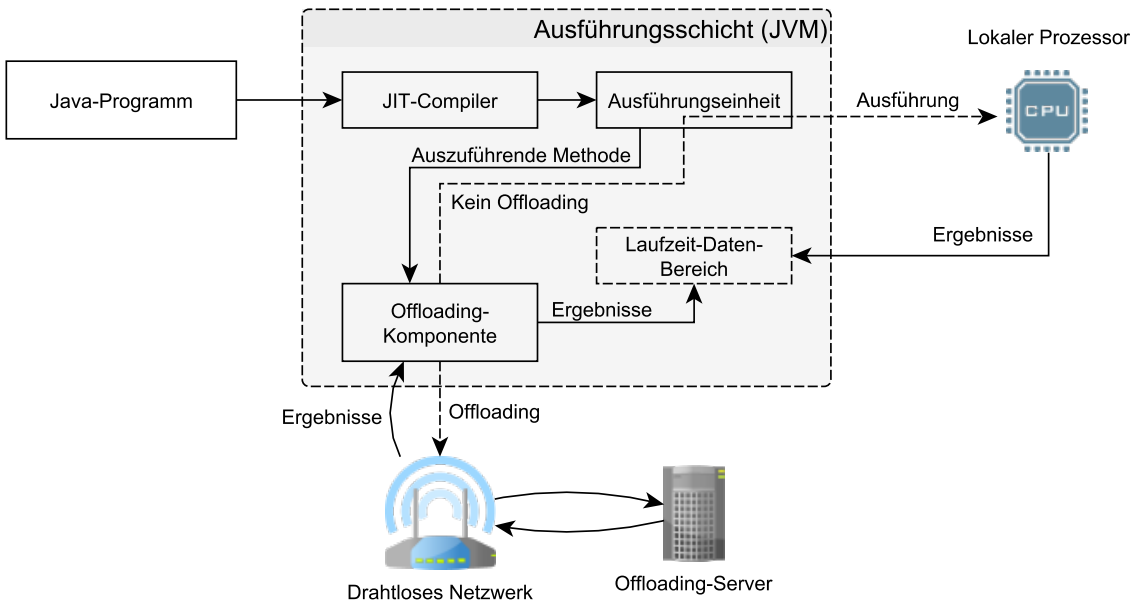


Abbildung 2.2.: Einordnung der Offloading-Komponente

ausgeführt werden soll. Errechnet die Offloading-Komponente dabei einen energetischen Vorteil, der bei einer Auslagerung entstehen würde, leitet sie die Übertragung der erforderlichen Daten zum Offloading-Server in der Cloud ein. Nachdem die entfernte Berechnung abgeschlossen ist, installiert sie das Ergebnis in die Laufzeitumgebung und die Ausführung kann normal fortgesetzt werden. In Abbildung 2.2 wird die Komponente innerhalb der JVM eingeordnet. Gestrichelte Pfeile stellen hierbei die beiden Optionen dar, für die sie sich entscheiden kann.

Die Auslagerung einzelner Methoden ist nicht immer sinnvoll, da die Menge an zu übertragenen Daten schnell nicht mehr im Verhältnis zur eingesparten Rechenzeit stehen kann. Da längere, abgeschlossene Berechnungen oft in eigenen Threads stattfinden, bietet es sich an, eher auf dieser Ebene Code-Offloading zu betreiben. Ein Beispiel für eine solche Realisierung ist CloneCloud [CIM+11]. Stellt die Software fest, dass es sinnvoll ist, eine Abfolge von Methoden nicht lokal auszuführen, wird dieser Teil der Anwendung so weit wie möglich auf einem Server berechnet. Das Ergebnis davon führt CloneCloud anschließend mit der lokalen Anwendung zusammen.

2.2.2. Energetische Analyse

Damit die Offloading-Komponente eine sinnvolle Entscheidung darüber treffen kann, ob ein Programmteil ausgelagert, oder lokal berechnet werden soll, muss es den energetischen Aufwand zum lokalen Berechnen einer Methode mit der Energie zum Übertragen und Empfangen der notwendigen Daten vergleichen. Die Energie zum Übertragen von Daten über eine mobile Netzwerkverbindung, wie sie bei Mobilgeräten vorgefunden wird, hängt entscheidend von der Paketumlaufzeit (engl. Round

Trip Time, RTT) ab [CBC+10]. Die RTT ist die Zeit, die ein Paket im Netzwerk von einer Quelle zum Ziel und zurück benötigt. Je nach Netzwerkanbindung variiert diese Zeit. Eine konkrete Messung der verbrauchten Energie beim Senden von Daten von einem Mobiltelefon wurde in [CBC+10] durchgeführt. In einem WLAN mit einer RTT von 25 ms verbraucht das Hochladen von 100 kB laut den Autoren 576 mJ. Bei derselben Verbindung und einer RTT von 50 ms erfordere die Aktion mit 989 mJ bereits fast die doppelte Energie. In einer mobilen Netzwerkverbindung über UMTS, bei der eine RTT von 220 ms gemessen wurde, würden sogar schon 2.762 mJ an Energie für das Hochladen von 100 kB fällig, schreiben die Verfasser der Publikation.
Neben der Netzwerkanbindung ist natürlich auch Wissen über die Größe der zu übertragenden Daten notwendig, um den Übertragungsaufwand errechnen zu können. Dazu bestimmt die Offloading-Komponente alle Objekte, welche von der Methode, die ausgelagert werden soll, verwendet werden. Die Autoren von [BDR14] setzen die gefundenen Objekte daraufhin mit ihrem Serialisierer in eine plattformunabhängige Form um. Die Größe dieses serialisierten Objekts kann nun für die Berechnung der Übertragungskosten genutzt werden.

Der energetische Aufwand zum lokalen Ausführen einer Methode wird hauptsächlich von der Zeit bestimmt, die diese bei der Ausführung auf dem lokalen Prozessor verbraucht. Ein einfacher Ansatz zum Finden dieser Größe ist das einfache Messen der Ausführungszeit von Methoden. Die Offloading-Komponente kann vor jeder Methodenausführung eine Messung deren Laufzeit starten, um beim nächsten Aufruf auf das Ergebnis davon zurückzugreifen. Je mehr Messdaten zur Verfügung stehen, desto genauer wird statistisch die Vorhersage der nächsten Ausführungsdauer einer Methode. Beim ersten Aufruf einer Methode liefert dieser Ansatz jedoch noch keine Daten. Dabei könnte bereits das Wissen über eine Minimaldauer die Offloading-Komponente dazu verleiten, einen Programmteil auszulagern. Dauert dieser länger, als berechnet, wird sogar mehr Energie gespart, als errechnet. Nur wenn die tatsächliche Dauer unter der berechneten liegt, kann durch Code-Offloading ein energetischer Nachteil entstehen. Das Finden der Ausführungszeit vor der ersten Ausführung einer Methode, nur auf Grundlage der zu diesem Zeitpunkt verfügbaren Daten, ist der zentrale Gegenstand dieser Arbeit.

2.2.3. Optimierungsproblem

Eine wichtige Aufgabe der Offloading-Komponente ist die Optimierung ihrer Arbeit. Sie soll dabei vor allem sicher stellen, dass keine Methoden ausgelagert werden, deren lokale Ausführung rentabler gewesen wäre, jedoch muss sie auch versuchen, durch das Auslagern möglichst viel Energie zu sparen und keine rentable Auslagerung auszulassen. Für eine optimale Arbeit der Komponente ist Wissen über die voraussichtliche Laufzeit der auszulagernden Programmteile nötig.

Anders als bei der Echtzeit-Programmierung ist hier jedoch eine Laufzeitvorhersage für den schlechtesten Fall unerwünscht bis kontraproduktiv: Geht die Offloading-Komponente davon aus, dass eine Methode längere Zeit dauert, als sie tatsächlich in Anspruch nimmt, entscheidet sie sich möglicherweise zur Auslagerung, obwohl die lokale Ausführung billiger gewesen wäre. Bei einer solchen Fehlentscheidung entsteht nicht nur auf dem Mobilgerät unnötiger Energieverbrauch, es werden auch kostbare Server-Ressourcen nutzlos verbraucht.
Nimmt die Komponente hingegen eine zu kurze Laufzeit an und schätzt somit die lokale Ausführung

für billiger ein, als sie in Wahrheit ist, entsteht lediglich der Energieverbrauch, den das Gerät auch ohne Code-Offloading gehabt hätte.

2.2.4. Mögliche Einsatzgebiete

Da Code-Offloading nicht nur Energie sparen, sondern auch die Ausführung von mobilen Anwendungen massiv beschleunigen kann, entstehen völlig neue Möglichkeiten auf Mobilgeräten. So macht es die Technologie beispielsweise möglich, auf Smartphones einen Echtzeit-Übersetzer für gesprochene Sprache zu betreiben und die Bildraten grafisch aufwändiger Spiele zu verdoppeln [CBC+10].

Sofern nicht in absehbarer Zukunft neue, deutlich leistungsfähigere Energiespeicher für Mobilgeräte zur Verfügung gestellt werden, kann Code-Offloading auf Dauer eine vielversprechende Technik sein, um sowohl längere Laufzeiten zu erzielen, als auch leistungsfähigere Anwendungen auf Smartphones, Tablets und anderen tragbaren Geräten bereitzustellen.

2.3. ASM

Wie im vorherigen Abschnitt gezeigt wurde, ist zum optimalen Treffen von Entscheidungen zum Auslagern von Teilen von Java-Programmen eine Vorhersage zur Ausführungsdauer des betreffenden Programmteils erforderlich. Dazu können die Java-Bytecodes der Anwendung analysiert werden. Zur Ausführung dieser Aufgabe gibt es das Werkzeug *ASM* [BLC02], welches die Arbeit auf der Ebene von Bytecodes erheblich vereinfacht.

Die Software ist eine Java-Programmbibliothek, welche in die eigene Java-Anwendung integriert werden kann. Sie basiert auf dem Entwurfsmuster *Besucher* (engl. visitor), welches die Möglichkeit zur Abkapselung von Funktionalitäten bietet. So hat die Lese-Komponente von ASM, der `ClassReader` eine `accept(...)`-Methode, welche mit einem Besucher-Objekt der Klasse `ClassVisitor` als Argument aufgerufen werden kann. Beim Einlesen von Daten ruft der `ClassReader` spezifische Methoden des Besuchers auf und informiert diesen dadurch über den Inhalt der eingelesenen Klasse. Zwei Besucher-Methoden eines `ClassVisitor`s sind:

- `void visit(int version, int access, String name, String signature, String superName, String[] interfaces)`
- `MethodVisitor visitMethod(int access, String name, String desc, String signature, String[] exceptions)`

Die Methode `visit(...)` wird aufgerufen, wenn eine Klasse im analysierten Java-Programm gefunden wird. Sie gibt dem Besucher-Objekt die Möglichkeit, sich die allgemeinen Informationen über die besuchte Klasse, die es als Parameter bekommt, abzuspeichern oder anderweitig zu verarbeiten. Die Methode `visitMethod(...)` hat darüber hinaus den Rückgabewert `MethodVisitor`. Das hier zurückgegebene Objekt wird vom `ClassReader` über weitere Details zur besuchten Methode informiert. Dazu implementiert ein `MethodVisitor` unter anderem:

- `void visitLabel(Label label)`

- `void visitFrame(int type, int nLocal, Object[] local, int nStack, Object[] stack)`
- `void visitInsn(int opcode)`
- `void visitIincInsn(int var, int increment)`
- `void visitIntInsn(int opcode, int operand)`
- `void visitLdcInsn(Object cst)`
- `void visitJumpInsn(int opcode, Label label)`

Durch diese Schnittstellen werden einem `MethodVisitor` alle relevanten Informationen über die Bytecode-Instruktionen in der besuchten Methode gezeigt: Der Aufruf von `visitLabel(...)` informiert darüber, dass die aktuelle Position im Code das Ziel einer Referenz an einer anderen Stelle sein kann, wie z. B. das einer Jump-Instruktion (S. 50). Einfache Instruktionen ohne Parameter bekommt der `MethodVisitor` über die Methode `visitInsn(...)` mitgeteilt. Erwartet eine Instruktion hingegen Parameter, werden dafür ausgelegte Methoden verwendet, wie etwa `visitIincInsn(...)` (für `iinc`), `visitIntInsn(...)` (für `bipush`, `sipush` und `newarray`) oder `visitLdcInsn(...)` (für `ldc`). So existiert für alle Parameter, die Java-Bytecodes haben können, eine solche Methode. Für einen `MethodVisitor` ist es durch diese einfach weiter zu verwendenden Informationen leicht, Daten einer Klasse, die nur im kompilierten Bytecode-Format vorliegt, zu verarbeiten.

Das Gegenstück zum `ClassReader` stellt der `ClassWriter` dar. Dieser funktioniert analog zum beschriebenen Verfahren, nur rufen in diesem Fall die Klassen der Anwendung aktiv die `visit...`-Methoden des `ClassReader`s auf. Wird dieser beispielsweise über eine Methode in einer Klasse informiert, erstellt er in der resultierenden *class*-Datei einen entsprechenden Eintrag.

3. Bestimmung der Ausführungszeit von Java-Bytecodes

In Kapitel 2 – *Grundlagen* ist gezeigt worden, dass es für das Treffen einer Entscheidung beim Offloading hilfreich ist, wenn vor der ersten Ausführung einer Methode eine Untergrenze ihrer Dauer bekannt ist. Zu diesem Zeitpunkt hat die Offloading-Komponente lediglich die Bytecode-Instruktionen, die den Ablauf der auszuführenden Methode beschreiben. Es ist offensichtlich notwendig, die Ausführungsdauer der einzelnen Instruktionen zu kennen, um aus diesem Wissen durch Aufsummieren die Ausführungsdauer einer Methode berechnen zu können.
Dieses Kapitel befasst sich damit, die Ausführungsdauer der kleinsten Einheiten von Java-Methoden, den Java-Bytecodes, zu messen und schafft damit eine Voraussetzung zur Vorhersage der Ausführungsdauer von Java-Programmen. Es werden die Probleme aufgezeigt, die es dabei zu lösen gilt, sowie eine Methode zur Implementierung vorgestellt.

3.1. Ansatz

Wie bereits in Kapitel 2 – Grundlagen gezeigt wurde, hat die Ausführung von Java-Bytecode-Instruktionen Auswirkungen auf den Operand-Stack im Laufzeit-Daten-Bereich der Java Virtual Machine. So erzeugt die Instruktion `iconst_0` einen konstanten Wert und legt ihn auf den Stack. Die JVM erwartet nun, dass dieser Wert von einer anderen Instruktion konsumiert wird. Bleibt das aus, kommt es im schlimmsten Fall zu undefiniertem Verhalten des Java-Programms oder zu einem Absturz der Virtual Machine. Einzelne Instruktionen können folglich nicht alleine gemessen werden, sondern brauchen stets eine oder mehrere weitere Instruktionen, die das Ergebnis der gemessenen Anweisung konsumieren oder deren Ausführung vorbereiten. Es ist dabei sinnvoll, Bytecode-Instruktionen in Produzenten und Konsumenten zu gruppieren. Ist die Ausführungsdauer von jeweils einer Instruktion dieser Gruppen bekannt, lässt sich die Ausführungsdauer jedes anderen Bytecodes ebenfalls bestimmen, indem eine unbekannte Instruktion mit einer oder mehreren bekannten produzierenden bzw. konsumierenden Instruktionen gemessen wird.

Um die Dauer der ersten produzierenden und konsumierenden Instruktionen festzustellen, ist eine Annahme erforderlich. Werden beide zusammen gemessen, wobei die eine Anweisung die Auswirkung der anderen neutralisiert, muss ein Verhältnis der Ausführungszeiten der zwei Instruktionen festgelegt werden, um die Messdauer auf die beiden einzelnen herunter rechnen zu können. Formell beschreibt diese Ausgangslage lediglich eine Annäherung. Die Aufgabe bei der Evaluation wird sein, ein möglichst realistisches Verhältnis zu finden.

Im Folgenden sei φ die Ausführungsdauer. Es wird nun die erste Annahme getroffen, dass `load`-Anweisungen gleich lange wie `store`-Anweisungen dauern. Für `iload` und `istore` lässt sich diese Annahme mit Gleichung 3.1 formulieren.

$$\varphi(iload) = \varphi(istore) = \frac{\varphi(iload, istore)}{2} \tag{3.1}$$

Nun ist die Ausführungsdauer einer konsumierenden und einer produzierenden Instruktion für den Datentyp *integer* bekannt. Mit diesem Wissen lassen sich weitere Zeiten berechnen:

$$\varphi(iconst) = \varphi(iconst, istore) - \varphi(istore)_{(3.1)} \tag{3.2}$$

$$\varphi(iadd) = \varphi(iload, iload, iadd, istore) - 2 * \varphi(iload)_{(3.1)} - \varphi(istore)_{(3.1)} \tag{3.3}$$

Insgesamt ist es so möglich, Instruktionen zu messen, die ausschließlich mit dem Datentyp *integer* arbeiten. Trifft man o. g. Annahme analog für je eine produzierende und eine konsumierende Instruktion der verbleibenden Datentypen, ist für alle Instruktionen die Dauer der notwendigen Vor- und Nachbereitungen für deren Ausführung bekannt, was diese messbar macht.

Bei einigen Anweisungen des Befehlssatzes einer JVM ist ein einfaches Messen jedoch nicht ausreichend, da ihre Ausführungszeit von der Eingabe abhängt. So dauert das Erstellen eines Arrays der Größe 100 mittels `newarray` sicherlich länger, als das Anlegen eines 10-elementigen Datenfeldes. Um die variable Dauer derartiger Instruktionen zu berücksichtigen, müssen mehrere Messungen mit unterschiedlichen Eingabewerten durchgeführt werden. Ihre Ergebnisse können zum Berechnen einer Annäherungsfunktion für die Ausführungsdauer im Verhältnis zur Eingabe verwendet werden.

Mit diesem Wissen sind die theoretischen Voraussetzungen für eine Messung der Ausführungsdauer von Java-Bytecodes geschaffen, sodass im nachfolgenden Abschnitt eine Möglichkeit zur Umsetzung dieser Theorie vorgestellt werden kann.

3.2. Implementierung

Um Messungen auf der niedersten Ebene der Sprache Java durchführen zu können, muss zunächst eine Möglichkeit gefunden werden, ein Java-Programm in Form von Java-Bytecodes zu erstellen. Das Schreiben von Quellcode scheidet aus, da der Java-Compiler nicht alle Konstruktionen erlaubt, die für das Vorhaben erforderlich sind. Mit ASM wurde bereits ein mächtiges Werkzeug zur Analyse und Modifikation auf Java-Bytecode-Ebene vorgestellt, welches ebenfalls hier zum Erstellen eines Programms genutzt werden kann, das Java-Bytecode-Instruktionen misst.

Der präziseste Zeitgeber eines Computers ist dessen Prozessor. Kein anderes Bauteil hat eine vergleichbar hohe Frequenz. Da Java-Bytecodes auf eben dieser Ebene ausgeführt werden, ist es so nicht möglich, einzelne Ausführungen von ihnen zu messen. Es wird notwendig, die zu messende Gruppe an Instruktionen wiederholt auszuführen, um rechnerisch die Dauer der kleinsten Einheiten zu bestimmen. Nur so kann eine ausreichende Messgenauigkeit erzielt werden.

Auf Programmebene wird dazu eine Schleife erstellt, die eine festgelegte Anzahl an Wiederholungen hat. Vor und nach ihrer Ausführung wird der Rückgabewert der nativen Java-Methode

`java.lang.System.nanoTime()` abgespeichert, um die Dauer der Schleife aus der Differenz der abgespeicherten Zeiten zu berechnen. Code-Auszug 3.1 zeigt die Bytecode-Anweisungen, mit denen die Mess-Schleife realisiert wurde.

Code-Auszug 3.1 Mess-Schleife in Java-Bytecodes

```
1   // Start-Zeit in Variable 0 ablegen
2   invokestatic(java/lang/System/nanoTime()J)
3   lstore_0
4
5   // Schleifenkopf
6   iconst_0
7   istore_2
8
9  l_start:
10  // Abbruchbedingung
11  iload_2
12  ldc(new Integer(1000000000))
13  if_icmpge(l_end)
14
15  // Schleifeninhalt.
16  // Hier werden zu messende Instruktionen eingefügt.
17
18  // Schleifenfuß
19  iinc(2, 1)
20  goto(l_start)
21
22 l_end:
23  // Ende-Zeit in Variable 2 ablegen
24  invokestatic(java/lang/System/nanoTime()J)
25  lstore_2
```

Da auch die Ausführung der Schleife Zeit kostet, die nicht zum endgültigen Messergebnis beitragen darf, muss diese überschüssige Zeit in einem ersten Schritt bestimmt werden. Dazu wird die Mess-Schleife leer ausgeführt. Hier und bei allen weiteren Messungen ist es wichtig, dass die zur Ausführung verwendete JVM keine Code-Optimierung durchführt. Leere Schleifen und Variablenzuweisungen, die später nicht mehr verwendet werden, würden durch Optimierungen aus dem Bytecode entfernt und die Messung wäre verfälscht.

Mit dem Werkzeug ASM kann nun eine Klasse gebaut werden, die sowohl eine Messung der durch die Mess-Schleife entstehende überschüssige Zeit vornimmt, als auch minimale Gruppen von zu messenden Anweisungen in die Mess-Schleife einfügt. Die generierte Klasse soll in einer Software verwendet werden können, welche die Messungen koordiniert und die Ergebnisse verarbeitet. Da Erzeugnisse von ASM dem Compiler, der die Software übersetzt, jedoch nicht bekannt sind, können in der Software Methoden, die in der generierten Klasse deklariert sind, nicht direkt angesprochen werden. Um dieses Problem zu lösen, wird in der Software eine Schnittstelle deklariert, welches die mittels ASM generierte Klasse implementiert. Das Produkt aus ASM kann somit in der Software als Implementierung von ihr angesprochen werden. Einige Methoden, welche die Schnittstelle vorschreibt, sind exemplarisch in Code-Auszug 3.2 dargestellt. Weitere Methoden erlauben die Messung der verbleibenden Bytecode-Instruktionen.

Code-Auszug 3.2 Schnittstelle zwischen generierter Mess-Klasse und der koordinierenden Software

```
1 public interface IBytecodeExperimentRunner {
2       public double iload_istore();
3       public double iconst_istore();
4       public double iload2_iadd_istore();
5       public double iinc();
6       public double iload2_if_icmpge();
7       public double iconst_tableswitch_goto();
8       ...
9 }
```

In der Methode `iload2_iadd_istore()` werden dabei zwei *integer*-Werte durch die doppelte Ausführung der Instruktion `iload` auf den Operand-Stack geladen, durch die Operation `iadd` addiert und das Ergebnis davon mittels `istore` vom Operand-Stack entfernt. Die Ergebnisse dieser Schnittstellen-Methoden sind die Ausführungszeiten der in ihren Namen angegebenen Instruktionen. Wird als erste Messung die Methode `iload_istore()` ausgeführt und das Ergebnis entsprechend der ersten Annahme für das Verhältnis von `iload` zu `istore` aufgeteilt, kann der danach bekannte Wert für die Ausführungszeit von `istore` vom Ergebnis der Methode `iconst_istore()` subtrahiert werden. Das Ergebnis der Subtraktion ist die Ausführungszeit der Instruktion `iconst`. Die auf diese Weise erzeugten Ergebnisse werden für eine spätere Verwendung bei der Vorhersage der Laufzeit von Methoden abgespeichert.

4. Analyse von Java-Bytecodes

Das vorangegangene Kapitel hat eine Vorbereitung für das Vorhersagen der Ausführungsdauer von Java-Programmen getroffen, indem es auf die Messung der kleinsten Einheiten von ihnen eingegangen ist. Nun ist die Dauer der Ausführung von Java-Bytecodes bekannt. Dieses Kapitel wird als Ziel das Finden der voraussichtlich ausgeführten Instruktionen einer Methode haben. Dabei wird erstens beleuchtet, wie durch statische Analyse von kompilierten Java-Methoden Strukturen mit festgelegter Wiederholungszahl erkannt werden. Darüber hinaus zeigen die Ausführungen in diesem Kapitel, wie die daraus entstandenen Ergebnisse durch dynamische Analyse zur Laufzeit verbessert werden können. Abschließend wird eine Möglichkeit zur Implementierung der beiden Techniken vorgestellt.

4.1. Ansatz

Im Ablauf von Methoden gibt es Gruppen von Bytecode-Instruktionen, die stets zusammenhängend ausgeführt werden. Diese werden als *Frames* bezeichnet und stellen atomare Einheiten einer Java-Methode dar. Es ist per Definition unter keinen Umständen möglich, dass die Methodenausführung innerhalb eines Frames abbricht oder zu einer anderen Stelle im Code springt.

Jedes Frame kann mehrere Vorgänger und Nachfolger haben. Erzeugt die letzte Instruktion in einem Frame eine bedingte Verzweigung (z. B. `if_icmpeq`, S. 51), so hat das Frame sowohl den Nachfolger, der im Programmablauf unmittelbar daran anschließt, als auch das Ziel der bedingten Verzweigung. Während der Programmausführung wird aufgrund der Eingabe, welche die Bedingungsabfrage erhält, über den zu gehenden Weg entschieden.
Eine solche Anordnung von Frames wird *Kontrollflussgraph* genannt. Dieser muss genau einen Knoten beinhalten, welcher keinen Vorgänger hat und somit der Einstiegspunkt ist. Weiterhin gilt als formale Bedingung für die Eigenschaft als Kontrollflussgraph, dass jeder andere Knoten vom Einstiegspunkt aus erreicht werden kann.

4.1.1. Statische Analyse

Offensichtlich gibt es nun mehrere Pfade, die durch einen Kontrollflussgraphen und damit auch durch ein Java-Programm führen. Entscheidend für die Laufzeit einer Methode sind vor allem Schleifen, die durch eine bedingte Verzweigung ein bestimmtes Codestück erneut ausführen. Das Ziel ist es daher, die Anzahl der Wiederholungen für Schleifen durch statische Analyse des Bytecodes einer Methode zu finden.

Oftmals haben Schleifen eine *Laufvariable*, die nach jedem Schleifendurchlauf erhöht oder verringert wird. Ohne eine solche Laufvariable lässt sich die Wiederholungsanzahl einer Schleife durch statische

Code-Auszug 4.1 Beispiel einer Schleife mit konstanter Wiederholungszahl.

```
1   // Laufvariable initiieren
2   iconst_0
3   istore_0
4   goto(schleifenfuß)
5  schleifenanfang:
6   // Schleifeninhalt
7   ...
8   // Laufvariable inkrementieren
9   iinc(0,1)
10 schleifenfuß:
11  // Abbruchbedingung prüfen
12  iload_0
13  bipush(100)
14  if_icmplt(schleifenanfang)
15  // (Weiterer Programmablauf)
```

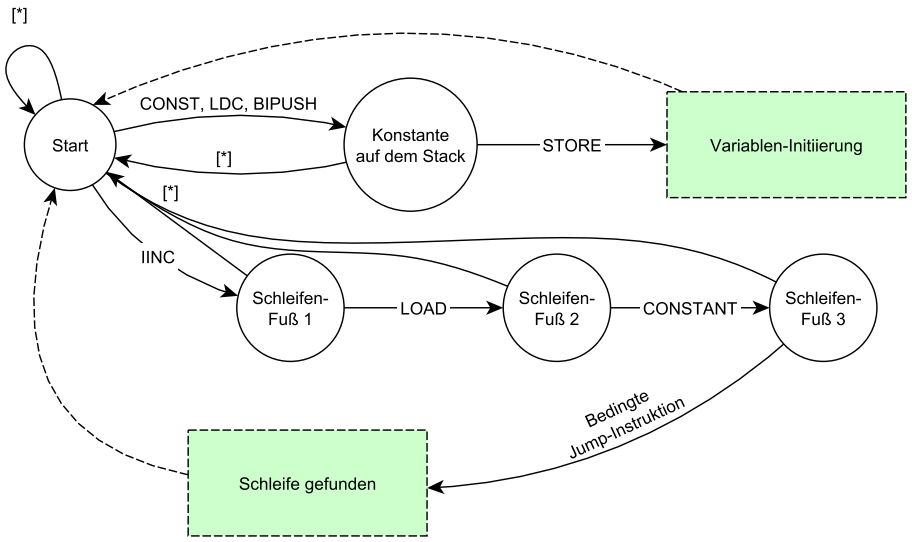


Abbildung 4.1.: Automat zur Erkennung von For-Schleifen mit festen Grenzen

Analyse im Allgemeinen nicht feststellen, da in diesem Fall keine direkt ablesbare, numerisch formulierte Bedingung für ihren Abbruch vorliegt. Unter Umständen kann durch zusätzlichen Aufwand in vom Inneren der Schleife aus aufgerufenen Methoden ein Hinweis auf die Anzahl der Schleifendurchläufe gefunden werden. Dies wird jedoch in der Minderheit der Fälle zutreffen, weshalb in dieser Arbeit keine solche weitere Untersuchung betrieben wird.

Im einfachsten Fall lässt sich die Anzahl der Schleifendurchläufe direkt aus dem Programmablauf ablesen, wie in Code-Auszug 5.1. Hier wird die Laufvariable mit dem Wert 0 initiiert und nach jedem Durchlauf um eins erhöht. Sobald ihr Wert größer oder gleich 100 ist, springt die Ausführung nicht mehr an den Anfang des Schleifeninhalts und die reguläre Programmausführung wird fortgesetzt.

Der Automat, welcher in Abbildung 4.1 dargestellt ist, veranschaulicht die möglichen Folgen an Instruktionen, die auf die Existenz von Schleifen mit einer konstanten Zahl an Wiederholungen hinweisen.
So erkennt er in einem ersten Schritt alle Variablen, die mit einem konstanten Wert initiiert werden. Jeder dieser Werte kann die Untergrenze einer Schleife sein und muss abgespeichert werden.
In einem zweiten Schritt erkennt der Automat einen Schleifenfuß. Hier muss zunächst eine der abgespeicherten Variablen mit konstantem Initial-Wert inkrementiert werden (durch die Instruktion `iinc`). Es handelt sich bei der inkrementierten Variable um die Laufvariable der Schleife (falls auch alle weiteren Bedingungen zutreffen und somit tatsächlich ein vollständiger Schleifenfuß vorliegt). Sollte das *Inkrement*, der Wert, welcher zur Laufvariable hinzugezählt wird, negativ sein, muss in diesem Fall die Untergrenze der Schleife einen größeren Wert haben, als die Obergrenze. Die Laufvariable zählt somit „rückwärts“. Wird eine nicht abgespeicherte Variable inkrementiert, liegt kein konstanter Initial-Wert vor und es kann keine Aussage über die Anzahl an Schleifendurchläufen getroffen werden, selbst wenn der Automat einen vollständigen Schleifenfuß erkennt.
Schließt daran das Laden derselben Variable (der Laufvariable) an, wird zusätzlich geprüft, ob die Instruktion hierfür das Ziel einer `goto`-Instruktion ist, welche früher im Programmablauf gefunden wurde. Wird daraufhin ein konstanter Wert erzeugt, die Obergrenze, und eine bedingte Verzweigung gefunden, die auf eine frühere Stelle im Code verweist, kann davon ausgegangen werden, dass der Fuß einer Schleife mit konstanten Grenzen der Laufvariable gefunden wurde. Trifft eine der genannten Bedingungen nicht auf die gefundenen Instruktionen innerhalb der analysierten Methode zu, wechselt der Automat in den Startzustand und beginnt die Analyse von der aktuellen Stelle an erneut. Die Anzahl an Schleifendurchläufen lässt sich nun wie folgt berechnen:

$$\frac{\text{Obergrenze} - \text{Untergrenze}}{\text{Inkrement}}$$

Neben Schleifen sind auch einfache bedingte Verzweigungen, welche durch `if`-Anweisungen im Quellcode eines Java-Programms hervorgerufen werden, für die Ausführungszeit einer Methode von Bedeutung. Es gibt grundsätzlich zwei Typen solcher Verzweigungen: Im einfachsten Fall wird durch eine Anweisung wie `if_icmpeq` (S. 51) darüber entschieden, ob ein Programmstück ausgeführt wird, oder nicht. Ist die Bedingung zur Ausführung des fragwürdigen Abschnitts nicht erfüllt, wird der Programmablauf mit der daran anschließenden Anweisung fortgesetzt. Darüber hinaus kann zweitens eine alternative Abfolge von Anweisungen festgelegt sein, welche anstelle des primären Abschnitts zur Ausführung gebracht wird. In jedem Fall kommt einer der beiden Abschnitte zur Ausführung. Die zwei Alternativen werden im Zusammenhang mit bedingten Anweisungen auch als Zweige bezeichnet. Ein beispielhaftes Programm in Form von Java-Bytecodes mit einer bedingten Verzweigung mit zwei Zweigen ist in Code-Auszug 4.2 zu finden.

Zur Erkennung derartiger Konstrukte dient der Automat, welcher in Abbildung 4.2 dargestellt ist. Findet dieser eine bedingte Verzweigung in Form einer Anweisung wie `if_icmp<op>` oder `if<op>` (S. 51), speichert er die von der Anweisung verwiesene Stelle, zu welcher bei erfüllter Bedingung gesprungen wird, ab. Gelangt er beim weiteren Durchlaufen des Programms zu einer abgespeicherten Position, war die bedingt verzweigende Anweisung ein Vorwärts-Verweis und es wird davon ausgegangen, dass eine `if`-Anweisung vorliegt. Ging der gespeicherten Position unmittelbar eine `goto`-Instruktion voraus, ist hingegen eine `if-else`-Anweisung gefunden worden, d. h. eine bedingte Verzweigung mit

Code-Auszug 4.2 Beispiel einer bedingten Verzweigung mit zwei Alternativen.

```
1   // Bedingungsabfrage
2   iload_0
3   ifge(else)
4  then:
5   // Zweig A
6   ...
7   goto(end)
8  else:
9   // Zweig B
10  ...
11 end:
12  // (Weiterer Programmablauf)
```

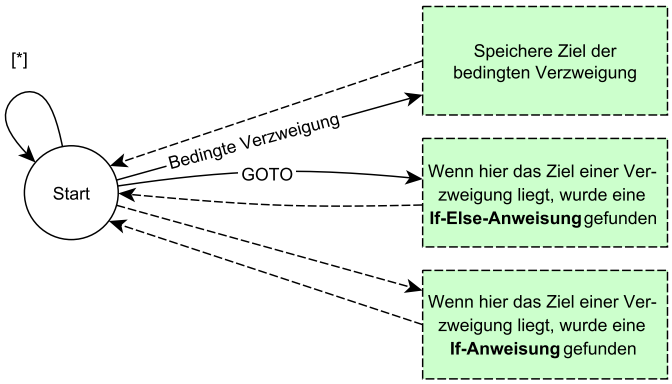


Abbildung 4.2.: Automat zur Erkennung von bedingten Verzweigungen

zwei Zweigen. Die `goto`-Instruktion dient dabei zum Überspringen des alternativen Zweiges nach der Ausführung des primären.

Nun ist für einige Schleifen die Wiederholungszahl bekannt und es können bedingte Verzweigungen in Methoden erkannt werden. Zusammen mit dem Ergebnis von Kapitel 3 – *Bestimmung der Ausführungszeit von Java-Bytecodes* kann nun die Zeit berechnet werden, welche bei der Ausführung von Methoden verbraucht wird. Als Resultat erhält man eine Vorhersage, welche von der Offloading-Komponente zum Treffen einer Entscheidung zum Auslagern oder lokalen Ausführen einer Methode verwendet werden kann.

4.1.2. Dynamische Analyse

Nach der statischen Analyse bleiben zahlreiche Teile von Methoden zurück, deren Ausführungsdauer aus Ermangelung an verfügbaren Informationen nicht vorhergesagt werden kann. Es bietet sich nun an, zur Laufzeit die Programmausführung zu verfolgen, um so mehr Daten für eine bessere Vorhersage zu erlangen.

In Kapitel 2 – Grundlagen ist gezeigt worden, dass das Werkzeug ASM den Kontrollflussgraphen einer Methode erzeugt. Die einzelnen Einheiten dieses Graphen sind Gruppen von Instruktionen mit der Eigenschaft, dass keine außer der letzten von ihnen eine Verzweigung der Programmausführung bewirkt und keine außer der ersten das Ziel einer solchen ist. Somit werden die Instruktionen innerhalb einer solchen Gruppe stets unmittelbar hintereinander ausgeführt und sind damit atomar. Diese Einheiten werden im Folgenden als *Frames* bezeichnet.

Die durch ASM gewonnenen Informationen werden nun dazu verwendet, an den Anfang eines jeden Frames einen Zähler in den Bytecode von Methoden einzusetzen, welcher bei jeder Ausführung eine lokale Variable inkrementiert. Am Ende einer Methode werden die entsprechenden Variablen ausgewertet. Damit ist für jede Instruktion der analysierten Methode die Anzahl deren tatsächlichen Ausführungen bekannt. Werden diese Daten gesammelt und mit den Parametern, mit denen die Methode aufgerufen wurde, in Verbindung gebracht, lässt sich die Vorhersagegenauigkeit der Ausführungsdauer einer Methode steigern.

4.2. Implementierung

Nun ist bekannt, wie mittels statischer und dynamischer Analyse Vorhersagen über die Ausführungsdauer von Java-Methoden getroffen werden können. Im Folgenden soll eine Möglichkeit der Implementierung dieser Ansätze erläutert werden.

4.2.1. Statische Analyse

Da zum Erkennen von Schleifen und bedingten Verzweigungen im Bytecode einer Methode keine Laufzeitinformationen notwendig sind, kann die statische Analyse direkt nach dem Kompilieren einer Anwendung durchgeführt werden. Das Ergebnis davon lässt sich im JAR-Archiv ablegen, um bei der Programmausführung für eine Offloading-Software direkt verfügbar zu sein. So muss ein Teil des analytischen Aufwands nicht auf dem Mobilgerät ausgeführt werden, was dem Ziel des Energiesparens zugute kommt.

Im Rahmen dieser Arbeit wurde ein Kommandozeilenprogramm entwickelt, welches bei Angabe eines JAR-Archivs die darin gespeicherten *class*-Dateien auf ablesbare Schleifengrenzen und einfache bedingte Verzweigungen untersucht und die Funde in einer Datei innerhalb des Archivs ablegt. Die Funktionsweise der Software wird in Abbildung 4.3 dargestellt.
Es besteht aus der Hauptkomponente `TimingAnalyzer`, welcher die Arbeit des ASM-`ClassReaders` anstößt und Ergebnisse sammelt. Als Besucher-Objekt für Klassen wird eine Instanz vom Typ `ClassAnalysisAdapter` eingesetzt, welcher die Besucher-Klasse `MethodAnalysisAdapter` für das Einlesen von Methoden zurückgibt. Letztere Klasse sammelt Daten zur besuchten Methode und meldet Ergebnisse an den `TimingAnalyser` zurück.
Die Klasse `MethodAnalysisAdapter` arbeitet zum Erzeugen einer internen Darstellung der eingelesenen Methoden mit einer Instanz des `MethodTimingResultBuilders` zusammen. Diese wird über die Eigenschaften der eingelesenen Methode informiert und baut mit diesen Informationen eine Liste mit Elementen vom Typ `InstructionFrame` auf. Jedes solche Element beinhaltet eine oder mehrere Bytecode-Instruktionen und kann mehrere Nachfolger, sowie einen Vorgänger haben. Hierbei wird

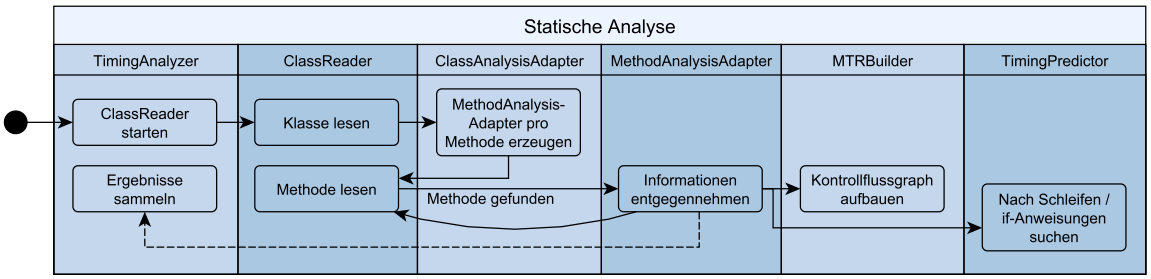


Abbildung 4.3.: Funktionsweise der Implementierung der statischen Analyse

der Kontrollflussgraph der Methode aufgebaut, wobei die Elemente der Liste Knoten im Graphen darstellen.

Genau wie der `MethodTimingResultBuilder` erhält auch eine Instanz des `TimingPredictors` die Daten, welche vom `ClassReader` produziert werden. Die erhaltenen Informationen nutzt er dabei, um den durch ihn repräsentierten Automaten, wie in Abbildungen 4.1 auf Seite 26 dargestellt, in andere Zustände zu überführen. Gerät er dabei in einen Endzustand (im Automat farblich markiert), führt er eine entsprechende Aktion durch. Stellt er so etwa eine Variable-Initiierung mit einem konstanten Wert fest, speichert er den gefundenen Initial-Wert für eine mögliche spätere Verwendung ab. Hat der `TimingPredictor` den Fuß einer Schleife gefunden, steht die Wiederholungszahl von dieser fest, welche er in alle `Instruktion-Frame`-Objekte schreibt, deren Instruktionen innerhalb der Schleife angeordnet sind. Hat eines der Frames dabei bereits eine voraussichtliche Ausführungszahl, wird diese mit der neu festgestellten multipliziert. Dieser Fall tritt bei verschachtelten Schleifen auf.
Mit derselben Vorgehensweise werden ebenfalls bedingte Verzweigungen im Programmablauf identifiziert. Hierbei werden jedoch die Zustandsübergänge des Automaten in Abbildung 4.2 auf Seite 28 verwendet. Beim Finden einer bedingten Verzweigung legt die Analysesoftware für sie eine eindeutige Nummer an, welche den, in ihren Zweigen befindlichen `InstructionFrame`-Objekten, mitgeteilt wird. Zudem erhalten die Objekte die Information, ob sie im primären oder alternativen Zweig angeordnet sind. Solche Informationen sind später zur Laufzeit verwertbar, um entscheiden zu können, welcher der Zweige die kürzere Ausführungszeit hat und somit in die Vorhersage miteinbezogen wird.

Ist die Analyse des `ClassReaders` beendet, werden die Ergebnisse, welche in Objekten vom Typ `TimingResult` vorliegen, in einer Datei abgelegt. Jedes Objekt, das abzuspeichernde Informationen beinhaltet, kann diese zu diesem Zweck in das JSON-Format [Cro06] überführen. Die entstehende Datei wird vom Analyseprogramm in das JAR-Archiv, welches analysiert wurde, integriert. So kann die Java Virtual Machine, welche das Java-Programm im Archiv ausführen möchte, direkt auf die Ergebnisse einer statischen Analyse zurückgreifen, ohne diese selbst produzieren zu müssen.

Minimierung von Frames

Objekte vom Typ `InstructionFrame` werden während der oben beschriebenen Analyse genau dann angelegt, wenn der ASM-`ClassReader` durch den Aufruf der Methode `visitLabel(...)`das Vorkommen

einer Markierung im Code anzeigt. So wird signalisiert, dass es sich bei der gegenwärtigen Stelle entweder um das Ziel einer Verzweigung handelt, oder dass sie aus anderen Gründen für den Kontrollfluss der Methode wichtig ist. Es werden dabei auch innerhalb von Anweisungsblöcken Markierungen angezeigt, die nicht Ziel oder Quelle einer Verzweigung sind. Für die dynamische Analyse ist es sinnvoll, so wenig Frames wie möglich anzulegen, da ansonsten, wie später gezeigt wird, unnötige Zähler in den Quellcode integriert werden. Aus diesem Grund wird nach dem Einlesen einer Methode für jedes `InstructionFrame`-Objekt geprüft, ob es nur genau einen Nachfolger hat. Ist dies der Fall, werden die Instruktionen des nachfolgenden Objekts zum geprüften hinzugezogen und dessen Nachfolger übernommen. Ein unnötiges Frame wird somit durch Zusammenführen mit einem anderen entfernt.

4.2.2. Dynamische Analyse

Für die Ausführung der dynamischen Analyse wurde im Rahmen dieser Arbeit die Jikes RVM (Research Virtual Machine) [jik] modifiziert, sodass sie vor dem Kompilieren einer Methode Zähler in deren Bytecode integriert und diese nach jeder Ausführung auswertet. Beim Einfügen von zusätzlichen Instruktionen ändern sich die Verweise von verzweigenden Anweisungen, weshalb die hier vorgestellte Implementierung vor dem Verändern des Ablaufs von Methoden eine interne Darstellung davon generiert, worin Verweise positionsunabhängig sind. Abbildung 4.4 zeigt eine Übersicht über die Klassen, welche am Installieren von Zählern beteiligt sind. Ihre Zusammenarbeit soll im Folgenden beschrieben werden.

Aufbau der internen Darstellung

Die Hauptkomponente, welche das Installieren von Zählern koordiniert, ist der `FrameCounterInstaller`. Wird er durch den Befehl `execute()` ausgeführt, erzeugt das Objekt aus dem bei seiner Instantiierung mitgegebenen Quell-Bytecode eine veränderbare Repräsentation in Form einer Instanz der Klasse `MoifiableBytecode`. Bei ihrer Initiierung setzt diese dazu den mitgegebenen Bytecode in eine entsprechende Liste an `Instruction`-Objekten um. Ein solches Objekt speichert dabei stets die Bytecode-Instruktion, welche es repräsentiert, sowie eine Liste an von der Instruktion erwarteten Parametern. Die Methode `getLength()` liefert die Länge der Anweisung plus die Anzahl ihrer Byte-Parameter und `getOffset()` summiert die Längen aller Vorgänger auf und berechnet so die Stelle der von ihr repräsentierten Instruktion im Bytecode. Außerdem hält ein `Instruction`-Objekt Zeiger zur nächsten und vorherigen Anweisung. Die aufgebaute Liste an solchen Objekten folgt somit dem Muster einer doppelt verketteten Liste.

Wird beim Durchlaufen des Bytecodes eine verzweigende Instruktionen `if_icmp<op>`, `if_acmp<op>`, `if<op>`, `goto`, `tableswitch` oder `lookupswitch` gefunden, erzeugt der `ModifiableBytecode` Objekte der entsprechenden Klasse, die von `Instruction` ableitet. Diese abgeleiteten Klassen definieren zusätzlich ein oder mehrere Ziele, bei welchen die Ausführung zur Laufzeit bei erfüllter Bedingung zum Verzweigen fortgesetzt wird. Ein `JumpInstruction`-Objekt hat somit genau ein Ziel, während ein `SwitchInstruction`-Objekt mehrere Ziele in Form eines Arrays beinhaltet, von denen zur Laufzeit eines abhängig vom Zustand des Operand-Stacks gewählt wird.
Im Bytecode sind die Verweise in Form einer `integer`-Zahl vorzufinden. Die Position des Ziels wird

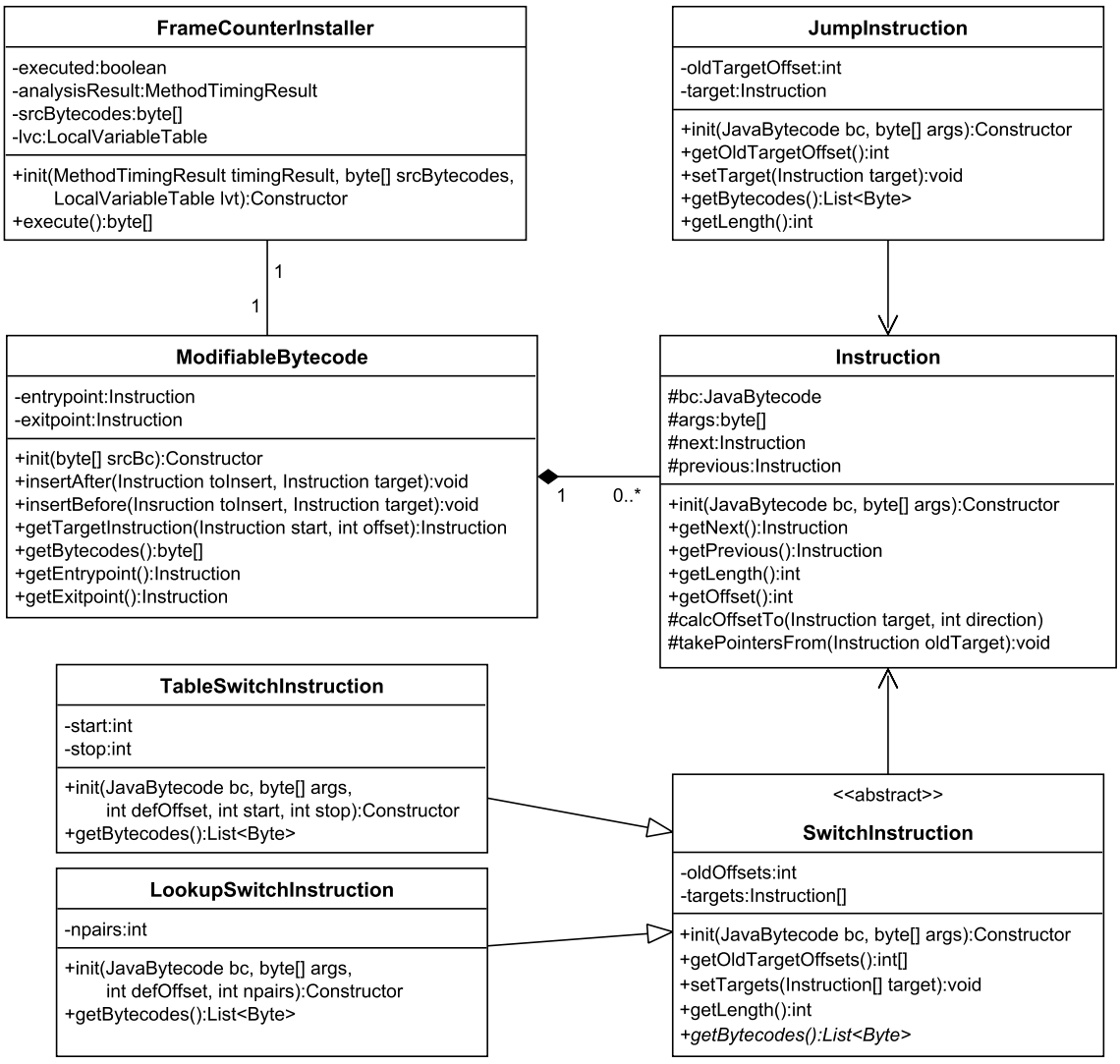


Abbildung 4.4.: Am Installieren von Zählern beteiligte Komponenten

durch Addition der Zahl zur Stelle der aktuellen Bytecode-Instruktion berechnet. Ist die Verweis-Zahl negativ, ist das Ziel an einer früheren Stelle im Programm zu finden.

Während dem Erstellen der Liste an `Instruction`-Objekten werden die Ziele in ihrer Ursprungsform in den Variablen `oldTargetOffset` in der `JumpInstruction` bzw. `oldOffsets` in der `SwitchInstruction` abgespeichert. Ist das Einlesen der Methode abgeschlossen, lassen sich die Offsets auflösen und die `Instruction`-Objekte, welche sich an den verwiesenen Stellen befinden, in die entsprechenden privaten Variablen der `Jump`- bzw. `SwitchInstruction`-Objekte speichern. Die von einem Offset verwiesene Instruktion kann gefunden werden, indem die Längen der Anweisungen, welche sich in der Richtung zum Ziel befinden, aufsummiert werden. Sobald die Summe dem Offset entspricht, ist das Ziel erreicht. Ist der Offset dabei positiv, wird von der verzweigenden Instruktion aus vorwärts gerichtet gesucht;

das nächste Element ist somit das in der Variablen `next` gespeicherte. Analog wird bei negativem Offset das nächste Element in der Variablen `previous` gesucht.

Installation der Zähler

Sind alle Verweise aufgelöst, kann mit dem Installieren der Zähler begonnen werden. Dabei greift der `FrameCounterInstaller` auf das Ergebnis der statischen Analyse zurück, in welchem der Kontrollflussgraph abgespeichert ist. Vor den Anfang eines jeden Frames schreibt die Komponente mittels der Methode `insertBefore(...)` im `ModifiableBytecode` eine `iinc`-Instruktion, welche eine in der `LocalVariableTable` neu angelegte Variable um eins erhöht. Damit diese Instruktion auch nach einer Verzweigung ausgeführt wird, müssen die verzweigenden Instruktionen, welche den Anfang des Frames als Ziel haben, nun auf die neue `iinc`-Anweisung zeigen. Für diese Veränderung wird die Methode `takePointersFrom(...)` des neuen `Instruction`-Objekts mit der ersten Instruktion des Frames als Parameter aufgerufen.

An den Anfang der Methode werden darüber hinaus die Initialisierungen der Zähler-Variablen mit dem Wert 0 eingefügt. Für diese Aktion wird die Instruktion `iconst_0` verwendet, welche auf den Operand-Stack der Methode den Wert 0 schreibt. Wurde der Stack der JVM mit der Größe 0 initiiert, muss diese vom `FrameCounterInstaller` erhöht werden, damit das Befüllen der Variablen fehlerfrei ausgeführt werden kann.

Auswertung

Da die Zähler-Variablen nicht Teil des ursprünglichen Java-Programms sind, kann auf sie nicht zugegriffen werden, wie Variablen normalerweise aus einem Objekt ausgelesen werden. Es ist notwendig, direkt auf den Laufzeit-Daten-Bereich der JVM zuzugreifen, welcher die Werte der Zähler beinhaltet.

Die Auswertung führt eine Methode innerhalb der modifizierten Jikes RVM durch. Diese wird an allen Ausstiegsstellen einer analysierten Methode durch ebenfalls eingefügte Bytecode-Instruktionen aufgerufen. Das Einfügen führt in diesem Fall der angepasste Compiler durch, indem er vor alle `return`- und `throw`-Instruktionen eine `invokestatic`-Anweisung in den resultierenden Maschinencode setzt. Somit wird ein regulärer Funktionsaufruf getätigt, bei dem die Virtual Machine definitionsgemäß ein neues Frame auf dem JVM-Stack erzeugt, welches die Daten der aufgerufenen Methode enthält. Indem diese auf den JVM-Stack zugreift, kann sie feststellen, von welcher Methode sie aufgerufen wurde, und die Zählerstände auslesen und verarbeiten.

5. Evaluation

In den vergangenen Kapiteln wurden Ansätze zur Analyse und Vorhersage der Ausführungsdauer von Java-Programmen vorgestellt. Die nachfolgenden Abschnitte sollen die Sinnhaftigkeit der gezeigten Methoden, sowie die Effizienz deren Implementierungsansätze aufzeigen.

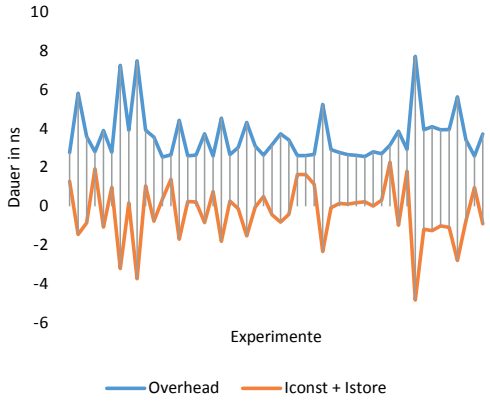
5.1. Messung der Dauer von Java-Bytecodes

Alle Messungen der Ausführungsdauer von Java-Bytecodes wurden auf dem Prozessor Intel® Core™ i7-4702HQ mit vier Kernen, acht Threads und einer maximalen Taktrate von 3,0 GHz durchgeführt. Bei anderen Hardware-Voraussetzungen ist es möglich, dass die Ergebnisse insgesamt schneller, langsamer, oder gar im Verhältnis unterschiedlich ausfallen. Diese Unterschiede hängen vor allem von der Leistungsfähigkeit und dem Befehlssatz des Prozessors ab. Bietet eine Recheneinheit z. B. für einige Operationen effizientere Maschinencode-Instruktionen an, ist es möglich, dass die betreffenden Java-Bytecodes, welche in solche umgesetzt werden, kürzere Ausführungszeiten haben. Dabei ist die Implementierung der JVM von Bedeutung, da diese das Umsetzen von Java-Bytecodes in Maschinencode-Instruktionen vornimmt. Somit kann offensichtlich auch sie Einfluss auf die Messergebnisse haben. Die Messungen in dieser Arbeit wurden auf der Jikes RVM [jik] durchgeführt, wobei diese so konfiguriert wurde, dass sie keine Optimierungen des Java-Bytecodes beim Kompilieren der Methoden durchführt.

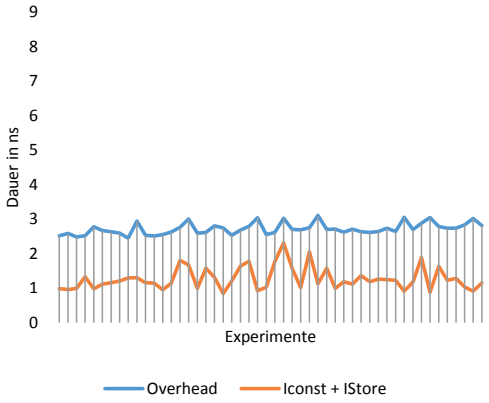
5.1.1. Genauigkeit durch Wiederholung

Das Ziel der Evaluation der Messung von Java-Bytecodes ist das Erzeugen von möglichst genauen Ergebnissen. Diese sind bedeutend, da die einzelnen Ausführungszeiten zum Berechnen der Dauer einer Methode aufsummiert werden. Im Bereich von Nanosekunden, in welchem die Ausführungsdauer von Maschinencode-Instruktionen auf dem Prozessor liegt, ist es leicht, eine Messungenauigkeit mit Faktor 3 und mehr zu erzielen. In den nachfolgenden Abschnitten soll erläutert werden, woraus eine solche Ungenauigkeit resultieren kann und wodurch es möglich ist, die Ergebnisse zu beeinflussen.

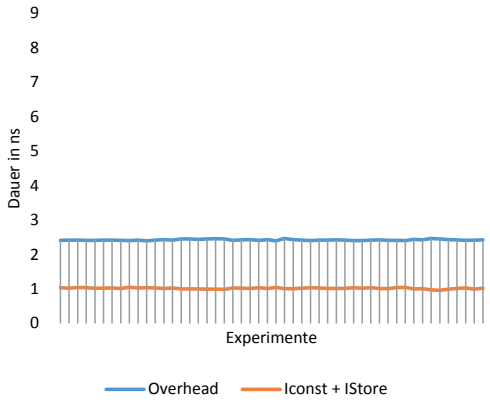
In Kapitel 3 wird gezeigt, wie Bytecode-Instruktionen für die Messung deren Ausführungsdauer in eine Mess-Schleife eingefügt werden. Ein Optimierungsproblem ist nun, eine Anzahl der Durchläufe dieser Schleife zu wählen. Ist diese Zahl zu hoch angesetzt, benötigt der Messvorgang mehr Zeit, als erforderlich wäre. Ist sie hingegen zu niedrig, werden die Ergebnisse der Messungen zu ungenau, um verwertbar zu sein. Um eine angemessene Anzahl an Durchläufen dieser Schleife zu finden, wird mit verschiedenen Konfigurationen experimentiert.



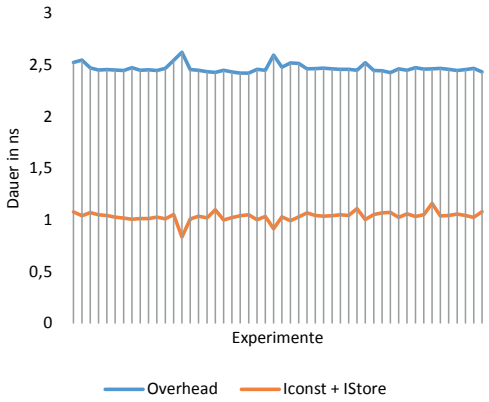
(a) 10^7 Ausführungen pro Experiment



(b) 10^7 Ausführungen pro Experiment bei voller Prozessorlast



(c) 10^9 Ausführungen pro Experiment bei voller Prozessorlast



(d) 10^9 Ausführungen pro Experiment bei voller Prozessorlast

Abbildung 5.1.: Durchschnittliche Dauer einer leeren Mess-Schleife

Abbildung 5.1 zeigt die Genauigkeit beim Messen mit unterschiedlichen Konfigurationen. Ein Diagramm stellt dabei die Ergebnisse einer Konfiguration dar, wobei bei der Messung von 5.1c dieselbe angewendet wurde, wie in 5.1d – zur Verdeutlichung der Messungenauigkeit wurde hier lediglich der Maßstab verändert. Es wurden pro Konfiguration 50 Experimente ausgeführt, welche die x-Achse der Diagramme befüllen. Das Ergebnis eines Experiments ist die durchschnittliche Dauer eines Schleifendurchlaufs, d. h. die gemessene Ausführungsdauer der Schleife mit allen ihren Durchläufen geteilt durch die Anzahl der Wiederholungen. Je weniger Abweichungen vom Mittelwert aller Ergebnisse es nun zu beobachten gibt, d. h. je gerader eine Linie ist, desto verwertbarer ist eine Konfiguration. Zu beachten gibt es dabei, dass eine Konfiguration nicht für jede Hardware gleich gut geeignet sein muss. Je nach Prozessortyp muss die Messung gegebenenfalls auf anderen Plattformen mit angepassten Parametern gestartet werden, um akzeptable Resultate hervorzubringen.

In den genannten Diagrammen sind jeweils zwei Datenreihen zu finden. Die Reihe „Overhead“ zeigt die durchschnittliche Dauer der leeren Mess-Schleife pro Experiment an, während die Schleife für das Ergebnis der Reihe „Iconst + Istore“ mit den Instruktionen `iconst` und `istore` befüllt wurde. Die Dauer der Ausführung der leeren Schleife wurde daraufhin von derjenigen der befüllten subtrahiert. So bleibt im Optimalfall genau die Dauer der gemessenen Instruktionen übrig.
In einer ersten Messung, deren Ergebnisse in Abbildung 5.1a zu finden ist, wurde die Mess-Schleife pro Experiment zehn Millionen mal ausgeführt. Anhand der Abbildung wird deutlich, dass die mittlere Dauer der leeren Schleife pro Experiment zwischen 3 ns und 8 ns schwankt. Durch die große Ungenauigkeit kommt es zu negativen Ergebnissen der Messung der Bytecode-Instruktionen. Die Abweichungen nach oben lassen sich durch die variable Taktzahl des Prozessors erklären. Ist dieser nicht stark ausgelastet, wird die Leistung der Recheneinheit zum Energiesparen gedrosselt, was die Ausführung von Maschinencode verlangsamt.
Aus diesem Grund wurde für ein zweites Experiment ein Java-Thread gestartet, welcher pausenlos Berechnungen auf dem Prozessor durchführt. Das veranlasst die Recheneinheit dazu, mit maximaler Taktrate zu operieren. Der Thread lastet dabei lediglich einen der acht Threads des eingesetzten Prozessors voll aus, was eine ungehinderte Durchführung von Messungen auf einem anderen Kern erlaubt. Wie die Ergebnisse in Abbildung 5.1b erkennen lassen, stellt sich eine deutlich höhere Messgenauigkeit ein. Diese liegt bei der leeren Mess-Schleife bei weniger als einer Nanosekunde, jedoch schwankt der Wert der Messung der Instruktionen noch gut um den Faktor Zwei.
Erst bei einer höheren Wiederholungszahl sind akzeptable Ergebnisse beobachtbar. Wie die Abbildungen 5.1c und 5.1d zeigen, gibt es bei einer Milliarde Wiederholungen der Mess-Schleife kaum noch Varianzen in den Messergebnissen.

Die höhere Genauigkeit hat jedoch ihren Preis: Während die Ausführung einer Schleife mit 10^7 Wiederholungen nur rund 10 Millisekunden an Zeit beansprucht, dauern 10^9 Schleifendurchläufe bereits mehrere Sekunden. Die Messung aller verfügbaren Java-Bytecode-Instruktionen mit dieser Konfiguration hat in einem Test auf derselben Maschine über zehn Minuten an Zeit verbraucht. Es muss folglich je nach Anwendungsfall entschieden werden, welches Maß an Genauigkeit erforderlich ist und wie sehr die Ausführungsdauer der Messung ins Gewicht fällt. Üblicherweise verändern sich die Messergebnisse auf einem Gerät jedoch nicht, was es erlaubt, solche Ergebnisse abzuspeichern und wiederzuverwenden.

Die Instruktionen aus dem Befehlssatz einer JVM haben teils sehr unterschiedliche Ausführungsdauern. So nimmt das Anlegen eines neuen Objekts mittels der Anweisung `new` ein Vielfaches der Zeit in Anspruch, die von einer Anweisung wie `iload` verbraucht wird. Bei solch langen Instruktionen macht es keinen Sinn, ihre Mess-Schleife ebenso oft auszuführen, wie diejenige von einfacheren Anweisungen. Aus diesem Grund werden die Instruktionen `newarray`, `anewarray` und `checkcast` bei ihrer Messung nur 50 Mio. mal wiederholt, `aastore` 80 Mio. mal sowie `multianewarray` und `new` eine Mio. mal. Damit stellt sich ein akzeptables Verhältnis zwischen Ausführungsdauer und Messgenauigkeit ein.

5.1.2. Gewichtung von Lade- und Speicher-Operationen

In Kapitel 3 wurde gezeigt, dass eine erste Annahme erforderlich ist, damit die Dauer aller Bytecode-Instruktionen bestimmt werden kann. Es ist notwendig, eine erste produzierende und eine konsumierende Instruktion gemeinsam zu messen, um die erhaltene Zeit gewichtet auf beide Anweisungen

aufzuteilen. Mit ihren dann bekannten Dauern als Konsument und Produzent lassen sie sich für die Messung weiterer Instruktionen verwenden. Als erster Produzent wird hier die Lade-Operation eines jeden Datentyps genommen und analog dazu die Speicher-Operation als Konsument. Für den Anfang wird hier vermutet, dass Lade- und Speicher-Operationen gleich viel Zeit in Anspruch nehmen.
In einem Experiment konnte für die Instruktionen `iload` und `istore` gemeinsam rund eine Nanosekunde an Zeit gemessen werden, d. h. mit der eben genannten Annahme dauert jede der beiden Instruktionen 0,5 ns. Einen Hinweis darauf, dass die gewählte Annahme nicht optimal ist, geben mit ihr berechnete Zeiten anderer Instruktionen, die negativ ausfallen. So stellte sich heraus, dass die Vergleichs-Operationen für den Datentyp *integer* `if_icmp<op>`, welche zusammen mit zwei `iload`-Instruktionen gemessen wurden, nach dem Abzug der vermuteten Dauer der Lade-Instruktion rund -0,6 ns dauert. Solche Funde geben Anlass zum Anpassen des gewählten Verhältnisses.

Durch weitere Experimente stellte sich heraus, dass ein Verhältnis von Lade- zu Speicher-Operationen von 1:10 deutlich weniger negative Werte verursacht. Dieses Verhältnis dient für die berechneten Zeiten in Tabelle 5.1 als Grundlage.

5.1.3. Limitierende Faktoren

Beim Ausführen von Messungen von Java-Bytecode-Instruktionen treten einige Probleme auf, welche deren Ergebnisse verfälschen können. So kann das Betriebssystem jederzeit die Ausführung des Programms unterbrechen, um einem anderen Prozess die Möglichkeit zur Abarbeitung seiner Aufgaben zu geben. Solche Unterbrechungen variieren abhängig von laufenden Programmen und Prozessorkapazität und können nicht vorhergesagt werden. Während den in dieser Arbeit durchgeführten Messungen wurde darauf geachtet, dass keine unnötigen Programme ausgeführt werden, damit die Ergebnisse so weitgehend wie möglich unbeeinflusst bleiben. Während der regulären Programmausführung kommt es jedoch ebenfalls zu derartigen Unterbrechungen, weshalb die Messungen nicht unrealistisch sind.
In Java kann darüber hinaus nicht nur das Betriebssystem in die Ausführung eines Programmes eingreifen, sondern auch der Garbage-Collector innerhalb der JVM. Da während den Messungen keine Objekte erzeugt werden, die von ihm beseitigt werden müssten, sind seine Auswirkungen auf die Ausführungszeit vernachlässigbar.

Dennoch wird die Ausführungsdauer in der Realität länger als gemessen ausfallen, denn bei der geringen Anzahl an stetig wiederholten Instruktionen müssen keine Daten aus dem Arbeitsspeicher oder gar dem Festspeicher nachgeladen werden. Alle notwendigen Informationen kann der Prozessor direkt aus seinem Cache beziehen. Die gemessenen Zeiten für die Ausführung von Instruktionen stellen somit einen Optimalfall dar, was jedoch zur Anforderung der Offloading-Komponente einer Minimalzeit für Methoden passt. Gegebenenfalls kann durch geeignete Experimente ein konstanter Faktor ermittelt werden, um den die Längen der Instruktionen angepasst werden müssen, damit sie den Ausführungszeiten der Realität nahe kommen. Dies wäre jedoch stark abhängig von der ausführenden Plattform.

aaload	1,02	dup_x1	0,51	iconst_4	0,10	l2i	0,73
aastore	99,31	dup_x2	0,83	iconst_5	0,10	ladd	5,13
aconst_null	0,58	dup2_x1	0,90	iconst_m1	0,10	laload	-0,32
aload	0,09	dup2_x2	0,96	idiv	2,41	land	5,83
aload_0	0,12	f2d	0,07	if_acmpeq	-0,62	lastore	0,74
aload_1	0,12	f2i	0,66	if_acmpne	-0,70	lcmp	1,12
aload_2	0,12	f2l	1,21	if_icmpeq	0,20	lconst_0	3,37
aload_3	0,12	fadd	0,71	if_icmpge	0,19	lconst_1	3,37
anewarray	423,42	faload	0,22	if_icmpgt	0,16	ldc	0,07
arraylength	0,16	fastore	0,10	if_icmple	0,15	ldc_w	0,07
astore	0,91	fcmpg	0,91	if_icmplt	0,15	ldc2_w	0,17
astore_0	1,22	fcmpl	0,96	if_icmpne	0,14	ldiv	16,73
astore_1	1,22	fconst_0	0,08	ifeq	-0,18	lload	0,16
astore_2	1,22	fconst_1	0,08	ifge	-0,19	lload_0	0.16
astore_3	1,22	fdiv	5,42	ifgt	-0,17	lload_1	0,16
baload	0,10	fload	0,09	ifle	-0,19	lload_2	0,16
bastore	0,12	fload_0	0,12	iflt	-0,17	lload_3	0,16
bipush	0,10	fload_1	0,12	ifne	-0,18	lmul	5,40
caload	0,09	fload_2	0,12	ifnonnull	-0,09	lneg	4,38
castore	0,25	fload_3	0,12	ifnull	-0,17	lookupswitch	0,02
checkcast	0,14	fmul	0,57	iinc	0,14	lor	5,57
d2f	0,95	fneg	0,76	**iload**	0,10	lrem	19,52
d2i	1,21	frem	8,18	**iload_0**	0,15	lshl	0,13
d2l	1,33	fstore	0,90	**iload_1**	0,15	lshr	4,56
dadd	1,76	fstore_0	1,24	**iload_2**	0,15	lstore	1,57
daload	0,70	fstore_1	1,24	**iload_3**	0,15	lstore_0	1,62
dastore	0,84	fstore_2	1,24	imul	0,92	lstore_1	1,62
dcmpg	1,56	fstore_3	1,24	ineg	0,72	lstore_2	1,62
dcmpl	1,56	fsub	0,65	instanceof	0,20	lstore_3	1,62
dconst_0	4,45	getfield	0,16	invokeinterface	3,19	lsub	5,20
dconst_1	4,45	getstatic	0,53	invokespecial	1,85	lushr	0,06
ddiv	1,45	goto	-0,08	invokestatic	1,66	lxor	6,21
dload	0,07	goto_w	-0,08	invokevirtual	1,98	monitorenter	95,54
dload_0	0,16	i2b	0,15	ior	1,04	monitorexit	0,00
dload_1	0,16	i2c	0,14	irem	2,45	**multianewarray**	4240,26
dload_2	0,16	i2d	0,30	ishl	2,11	**new**	408,97
dload_3	0,16	i2f	0,60	ishr	2,12	**newarray**	440,59
dmul	1,51	i2l	3,88	**istore**	0,97	nop	0,00
dneg	4,67	i2s	0,42	**istore_0**	1,48	pop	-0,15
drem	5,60	iadd	1,15	**istore_1**	1,48	pop2	0,10
dstore	0,71	iaload	0,09	**istore_2**	1,48	putfield	0,00
dstore_0	1,60	iand	1,01	**istore_3**	1,48	putstatic	-0,11
dstore_1	1,60	iastore	0,07	isub	0,94	saload	0,10
dstore_2	1,60	iconst_0	0,10	iushr	2,10	sastore	0,11
dstore_3	1,60	iconst_1	0,10	ixor	1,13	sipush	0,08
dsub	1,57	iconst_2	0,10	l2d	0,20	swap	0,27
dup	0,50	iconst_3	0,10	l2f	0,67	tableswitch	1,38

Tabelle 5.1.: Ergebnise der Messung von Bytecode-Instruktionen in Nanosekunden

5.1.4. Ergebnisse

Tabelle 5.1 zeigt die Ergebnisse der Messung der einzelnen Instruktionen. Wichtige Instruktionen sind in ihr durch Fettdruck markiert, wie etwa die Lade- und Speicher-Instruktionen für den Datentyp *integer*, welche bei der Messung zahlreicher anderer Instruktionen für Vor- und Nachbereitungen verwendet werden, sowie die Bytecode-Instruktionen zum Erzeugen von Arrays und Objekten, welche eine besonders hohe Dauer aufweisen.

Auffällig sind die gemessenen Zeiten für die Lade- und Speicher-Operationen. Hier sind Instruktionen der Form `iload_`*n* (S. 45) langsamer als solche, die einen Parameter erwarten, in welchem die geladene oder gespeicherte Variable angegeben wird, wie `iload`. Erwartungsgemäß müssten die beiden Typen an Instruktionen gleichschnell ablaufen, da sie von der Jikes RVM in dieselben Maschinencode-Instruktionen umgesetzt werden. In dieser Arbeit kann keine weitere Analyse dieser Besonderheit unternommen werden und soll stattdessen künftige Untersuchungen anregen.

5.2. Statische Analyse

Um festzustellen, wie präzise die Vorhersage der statischen Analyse ist, wurde die Anzahl an Ausführungen der einzelnen Frames von Testmethoden gezählt. Tatsächlich sagt die modifizierte JVM die Häufigkeit der Ausführungen für den Schleifenrumpf einer `for`-Schleife, wie in Code-Auszug 5.1 dargestellt, korrekt vorher. Eine Vorhersage ist jedoch nur dann möglich, wenn eine Schleife tatsächlich konstante Schleifengrenzen aufweist. Wird in der Abbruchbedingung eine lokale Variable zum Vergleich herangezogen, oder erhält die Laufvariable keinen konstanten Initialwert, kann keine Wiederholungszahl berechnet werden.

Code-Auszug 5.1 Eine For-Schleife mit konstanter Grenze.

```
1 for (int i = 1; i < 1000000; i++) {
2 ...
3 }
```

Wie in Kapitel 4 – *Analyse von Java-Bytecodes* gezeigt wurde, erkennt das Analyseprogramm Schleifen mittels eines Automaten, welcher durch bestimmte Muster im Methodenablauf in einen Endzustand wechselt. Je komplexer der Automat ist, desto mehr unterschiedliche Typen von Schleifen können durch ihn erkannt werden. Möchte man seine Erkennungsrate vergrößern, muss er so erweitert werden, dass er die Konstrukte, welche von Java-Compilern bei allen möglichen Arten von Schleifen produziert werden, erkennt. Konstrukte im Bytecode von Methoden, zu denen keine Informationen zur Häufigkeit der Ausführung auffindbar sind, die nicht vom Automaten erkannt werden, zählen bei der hier zugrunde liegenden Implementierung nicht zur Ausführungszeit von Methoden dazu. Dadurch gibt es Abweichungen der vorhergesagten Laufzeit nach unten, jedoch werden keine zu hohen Ausführungszeiten berechnet. So werden kontraproduktive Entscheidungen der Offloading-Komponente vermieden.

Für die Evaluation der Erkennung von `if`-Anweisungen wurde das im Rahmen dieser Arbeit erstellte Kommandozeilenprogramm mit einem Testprogramm als Eingabe ausgeführt. Dieses beinhaltete

sowohl einfache bedingte Anweisungen, als auch solche mit alternativen Zweigen, in welchen weitere Bedingungen abgefragt wurden. Die Aufgabe des Programms ist das Identifizieren von Abfolgen von Bytecode-Instruktionen, welche durch eine `if`-Anweisung im Quellcode erzeugt wurden. In der Ausgabedatei fanden sich die erwarteten Informationen zu den vorhandenen Abfragen und die Implementierung des Automaten ist somit korrekt.

Es bleibt die Frage offen, ob es nicht sinnvoller wäre, die statische Analyse innerhalb der Java Virtual Machine auszuführen, wo sich auch die Offloading-Komponente befindet und wo es möglich ist, dass bereits Laufzeit-Informationen zur analysierten Methode verfügbar sind. Mit solchen Laufzeit-Daten stehen womöglich variable Grenzen von Schleifen bereits fest. Durch den Rechenaufwand, der bei der Analyse anfällt, ist es jedoch sinnvoll, Untersuchungen so weit als möglich im Vorhinein durchzuführen. Man könnte die Variablen, deren Werte über die Wiederholungszahl einer Schleife entscheiden, in der Ergebnis-Datei innerhalb des JAR-Archivs ablegen, um zur Laufzeit ohne erneute statische Analyse Werte solcher Variablen zur Vorhersage der Ausführungszeit zu verwenden. So würden die Vorteile des Energiesparens beim statischen Analysieren außerhalb des mobilen Gerätes und der Möglichkeit der Verwendung von lokalen Variablen zur Laufzeit geltend.

5.3. Dynamische Analyse

Zur Evaluation der Installation von Zählern diente ein Testprogramm, welches aus drei Methoden besteht, bei denen jeweils auf unterschiedliche Anweisungskonstrukte Wert gelegt wurde.

Im ersten Fall führt eine Methode nur Berechnungen mit einer *double*-Variablen aus. Dazu wurde in einer For-Schleife, welche $5 * 10^8$ mal wiederholt, eine Switch-Anweisung platziert, die der Compiler als `lookupswitch` umsetzt. Innerhalb der Schleife finden sich vier Frames, d. h. dort werden vier Zähler installiert.
In einer zweite Methode wird in einer 10^6 mal ausgeführten Schleife zusätzlich zur Switch-Anweisung und arithmetischen Operationen eine `HashMap` aufgebaut. Dabei prüft das Programm vor allem bei jedem Schleifendurchlauf ab, ob ein Wert in der `HashMap` existiert, was in einem Funktionsaufruf an eine Methode der Java-API (siehe S. 12) resultiert. Ein Teil der Laufzeit läuft somit außerhalb der Funktion ab und ist nicht Teil der dynamischen Analyse.
Den dritten Fall stellt eine rekursive Methode dar, welche in jeder Rekursion arithmetische Operationen mit einer *double*-Variablen durchführt.

Jede der drei Test-Methoden wurde mit und ohne Installation, Ausführung und Auswertung von Zählern am Anfang von Frames gemessen, um deren Auswirkung auf die Ausführungsdauer festzustellen. Während bei Fall zwei und drei kein eindeutiger Unterschied feststellbar ist, fällt das Ergebnis der Messungen bei der Testmethode, welche nur Berechnungen durchführt, deutlich aus: Im Mittel über 20 Ausführungen dauert eine Funktionsausführung ohne Zähler 6,4 Sekunden und mit aktivierter dynamischer Analyse bereits 7,0 Sekunden. Die Ausführung der Instruktionen, welche die Zähl-Variablen erhöhen, nehmen somit gut 10 % der regulären Ausführungszeit in Anspruch.
Diese Größe hängt einerseits von der Länge der zusammenhängenden Anweisungen, d. h. der Frames ab. Je mehr Instruktionen durch einen Zähler abgedeckt werden können, desto seltener muss eine Variable erhöht werden. Andererseits entscheidet das Verhältnis der Länge der Instruktionen in der Methode zu den `iinc`-Anweisungen der Zähler über die verhältnismäßige zeitliche Auswirkung

der dynamischen Analyse. Beinhaltet eine Methode zahlreiche zeitintensive Anweisungen wie `new`, `newarray` oder auch Methodenaufrufe, fällt das Zählen der Frames kaum mehr ins Gewicht.

6. Zusammenfassung und Ausblick

In den vergangenen Kapiteln wurde eine Möglichkeit vorgestellt, Code-Offloading durch eine bessere Entscheidungsgrundlage effizienter zu gestalten. So kann die Offloading-Komponente mit den Ergebnissen dieser Arbeit Informationen über die Ausführungsdauer von Java-Methoden generieren, um so eine Einschätzung über die Sinnhaftigkeit der Auslagerung von Programmteilen auf entfernte Server zu treffen.
Hierzu wird die Eigenheit der Programmiersprache Java ausgenutzt, dass ihr Quellcode nicht zu Maschinencode kompiliert wird, sondern zu Java-Bytecodes. In dieser Form liegen der Offloading-Komponente Methoden vor, in welchen durch die statische Analyse (Kapitel 4) Schleifen und bedingte Verzweigungen erkannt werden können. So entsteht Wissen über die Häufigkeit der Ausführungen einzelner Programmteile. Zusammen mit den Messungen der Ausführungsdauer von Java-Bytecodes, welche in Kapitel 3 beschrieben wurden, lassen sich diese Häufigkeiten auf die Gesamtdauer einer Methode hochrechnen. Die Erkennungsrate von Schleifen und anderen verzweigenden Konstrukten, welche Auswirkungen auf die Ausführungszeit einer Methode haben, hängt dabei von der Komplexität des Automaten ab, welcher für diese Aufgabe zugrunde liegt. Verschiedene Java-Compiler erzeugen potentiell für denselben Quellcode unterschiedliche Bytecode-Konstrukte. Analysiert man die unterschiedlichen Ausgaben, lässt sich der Automat zur Schleifenerkennung entsprechend erweitern und die Analyse-Ergebnisse werden insgesamt genauer.
Die Dynamische Analyse macht es möglich, die Abschätzung der statischen Analyse zur Laufzeit zu präzisieren, um eine genauere Vorhersage der Ausführungsdauer von Methoden treffen zu können. Dabei wird durch das Einfügen neuer Zähl-Anweisungen die Häufigkeit der Ausführung der größten zusammenhängenden Gruppen von Bytecode-Instruktionen ermittelt. Die gewonnenen Ergebnisse helfen, die durch fehlendes Wissen bedingten Ungenauigkeiten der statischen Analyse zu begleichen. Die Möglichkeit, zur Laufzeit Methodenparameter bei der Vorhersage miteinzubeziehen, macht diese Technik derjenigen überlegen, welche nur statistische Hochrechnungen auf Grundlage von gemessenen Zeiten verwendet.
Die Evaluation in Kapitel 5 zeigt auf, dass Messungen der Bytecode-Instruktionen nur bei großer Wiederholungszahl und vollständiger CPU-Auslastung brauchbare Ergebnisse liefern. Zudem ist es notwendig, eine Annahme über das Verhältnis der Ausführungsdauern zweier Instruktionen zu treffen, da eine Anweisung stets eine Auswirkung auf den Operand-Stack hat, welche durch eine nachfolgende wieder neutralisiert werden muss. Alle weiteren Messungen hängen von dieser ersten Annahme ab. Es ist somit wichtig, dass sie sorgfältig gewählt und evaluiert wird.

Abschließend lässt sich festhalten, dass mit der Bestimmung der Ausführungszeit von Java-Anwendungen zur Laufzeit eine potentiell gewinnbringende Methode gefunden wurde, um Code-Offloading effizienter zu gestalten. Mit den Ergebnissen aus dieser Arbeit kann eine Offloading-Komponente aufgrund fundierter Daten entscheiden, ob ein Programmteil ausgelagert oder lokal

berechnet werden soll. Vielleicht bringt uns das einen Schritt näher zu Mobilgeräten, deren Energiespeicher nicht täglich geladen werden müssen.

Ausblick

Für die Erweiterung und Verfeinerung der Ergebnisse dieser Arbeit bleiben einige Ansätze für weitere Forschung offen. So kann untersucht werden, wie sich die Messung der Ausführungsdauer der Bytecode-Instruktionen beschleunigen und präzisieren lässt. Möglich ist hier unter Umständen eine Betrachtung der von der JVM erzeugten Maschinenbefehle, um unter Einbeziehung der Prozessorgeschwindigkeit genaue Daten zur Dauer von Methoden zur erhalten. Weiter kann erforscht werden, ob eine andere Wahl der Gewichtung von Lade- und Speicher-Anweisungen die übrigen Ergebnisse verbessert, oder ob sich gar andere produzierende und konsumierende Instruktionen besser als Basis für weitere Messungen eignen.
Die statische Analyse lässt sich durch einen komplexeren Automaten verfeinern, welcher mehr Bytecode-Konstrukte erkennt. So wird für mehr Programmteile die Ausführungshäufigkeit bekannt und die Vorhersagegenauigkeit der Ausführungszeit steigt. Noch exaktere Vorhersagen sind über das Miteinbeziehen der Ergebnisse der dynamischen in die statische Analyse möglich. Mit dem Wissen der Häufigkeit der Ausführung von Teilen einer Methode können Rückschlüsse über das Verhältnis der Ausführungsdauern einzelner Bytecode-Instruktionen getroffen und so künftige Vorhersagen verfeinert werden.

A. Anhang

A.1. Java-Bytecodes

Im Folgenden werden die verfügbaren Java-Bytecodes mit deren Auswirkungen auf den Operand-Stack im Laufzeit-Daten-Bereich einer Java Virtual Machine vorgestellt. Für detailliertere Informationen über den Befehlssatz einer JVM kann deren Spezifikation konsultiert werden [LY99].

A.1.1. Primitive Datentypen

Laden und Speichern

Für die Datentypen *integer*, *long*, *double* und *float* existieren jeweils Bytecode-Instruktionen zum Laden aus, sowie zum Speichern in die lokalen Variablen einer Methode. Als Beispiel werden hier die Instruktionen für den Datentyp *integer* angeführt. Durch Ersetzen des ersten Buchstabens der Namen der Instruktionen durch *l* (für *long*), *d* (für *double*) oder *f* (für *float*) werden die entsprechenden Instruktionen für den jeweiligen Datentyp erreicht.

iload_<n> *Keine Argumente.*
Lädt die n-te lokale Variable auf den Operand-Stack, wobei diese den Typ *integer* hat.
Für $0 \leq n \leq 3$.

iload *Gefolgt von einem Index-Byte* n.
Lädt die n-te lokale Variable auf den Operand-Stack, wobei diese den Typ *integer* hat.

istore_<n> *Keine Argumente.*
Speichert den obersten Wert vom Operand-Stack in der n-ten lokalen Variable und entfernt den Wert vom Operand-Stack, wobei die Variable den Typ *integer* hat.
Für $0 \leq n \leq 3$.

istore *Gefolgt von einem Index-Byte* n.
Speichert den obersten Wert vom Operand-Stack in der n-ten lokalen Variable und entfernt den Wert vom Operand-Stack, wobei die Variable den Typ *integer* hat.

Konstante Werte erzeugen

Benötigt eine Methode für eine Berechnung keinen Wert aus einer lokalen Variable, sondern eine konstante Zahl, kann sie diese durch eine der folgenden Instruktionen erzeugen.

iconst_<n> *Keine Argumente.*
Konstruiert den *integer*-Wert n und schreibt ihn an die oberste Stelle des Operand-Stacks.
Für $0 \leq n \leq 5$.

iconst_m1 *Keine Argumente.*
Konstruiert den Integer-Wert -1 und schreibt ihn an die oberste Stelle des Operand-Stacks.

lconst_<n> *Keine Argumente.*
Konstruiert den *long*-Wert n und schreibt ihn an die oberste Stelle des Operand-Stacks.
Für $n \in \{0, 1\}$.

dconst_<n> *Keine Argumente.*
Konstruiert den *double*-Wert n und schreibt ihn an die oberste Stelle des Operand-Stacks.
Für $n \in \{0, 1\}$.

fconst_<n> *Keine Argumente.*
Konstruiert den *float*-Wert n und schreibt ihn an die oberste Stelle des Operand-Stacks.
Für $n \in \{0, 1, 2\}$.

Arithmetische Operationen

Für die Datentypen *integer*, *long*, *double* und *float* existieren jeweils äquivalente Bytecode-Instruktionen für arithmetische Operationen. Eine Ausnahme stellt die Instruktion *iinc* dar. Sie kann nur auf *integer*-Variablen angewendet werden. Stellvertretend werden im Folgenden die Instruktionen für den Datentyp *integer* vorgestellt.

iadd *Keine Argumente.*
Entfernt die beiden obersten Werte $w1$ und $w2$ vom Operand-Stack, wobei diese den Typ *integer* haben müssen, und schreibt das Ergebnis der **Addition** der beiden Werte an die oberste Stelle des Operand-Stacks.

isub *Keine Argumente.*
Entfernt die beiden obersten Werte $w1$ und $w2$ vom Operand-Stack, wobei diese den Typ *integer* haben müssen, und schreibt das Ergebnis der **Subtraktion** der beiden Werte an die oberste Stelle des Operand-Stacks.

imul *Keine Argumente.*
Entfernt die beiden obersten Werte $w1$ und $w2$ vom Operand-Stack, wobei diese den Typ *integer* haben müssen, und schreibt das Ergebnis der **Multiplikation** der beiden Werte an die oberste Stelle des Operand-Stacks.

idiv *Keine Argumente.*
Entfernt die beiden obersten Werte $w1$ und $w2$ vom Operand-Stack, wobei diese den Typ *integer* haben müssen, und schreibt das Ergebnis der **Division** der beiden Werte an die oberste Stelle des Operand-Stacks.

irem *Keine Argumente.*
Entfernt die beiden obersten Werte $w1$ und $w2$ vom Operand-Stack, wobei diese den Typ *integer* haben müssen, und schreibt den **Rest** einer Division der beiden Werte an die oberste Stelle des Operand-Stacks. Der Rest wird wie folgt berechnet.

$$result = w1 - \frac{w1}{w2} * w2$$

ineg *Keine Argumente.*
Negiert den obersten Wert des Operand-Stacks, wobei dieser den Typ *integer* haben muss.

iinc *Gefolgt ein einem vorzeichenlosen Byte $index$ und einem vorzeichenbehafteten Byte $const$.*
Erhöht die lokale Variable an der Position $index$, die vom Typ *integer* sein muss, um den Wert $const$.

Bit-Operationen

Für die ganzzahligen Datentypen *integer* und *long* stehen weiterhin Operationen auf Bit-Ebene zur Verfügung. Diese sind

- die arithmetische Verschiebung nach links (`ishl`, `lshl`) und rechts (`ishr`, `lshr`),
- das bitweise ODER (`ior`, `lor`),
- das bitweise UND (`iand`, `land`), sowie
- das bitweise exklusive ODER (`ixor`, `lxor`).

Konvertierungen

Im Befehlssatz einer JVM sind ebenfalls Instruktionen zu finden, die Werte verschiedenen Typs ineinander umwandeln. So ist es möglich, durch Instruktionen, wie *i2d* oder *f2l* Variablen von den Datentypen *integer*, *long*, *double* und *float* ineinander zu konvertieren. Außerdem können *integer*-Werte in *byte*-, *char*- und *short*-Werte umgewandelt werden.

A.1.2. Komplexe Datenstrukturen

Referenzen

Mit Objekten wird in der JVM nicht direkt auf dem Operand-Stack gearbeitet. Sie lagern stattdessen im Heap und werden über ihre Position adressiert. Verweise zu Objekten werden im Java-Bytecode genau wie Variablen behandelt. Analog zu den primitiven Datentypen kann mit den Instruktionen `aload_<n>` und `aload` ein Verweis aus einer lokalen Variable geladen werden, sowie durch `astore_<n>` und `astore` ein Verweis vom Operand-Stack in einer lokalen Variable abgelegt werden. Weiterhin ist es möglich, mit der Instruktion `aconst_null` einen Verweis zum *null*-Objekt auf dem Operand-Stack zu erstellen.

Objekte und Felder

Neue Instanzen von Klassen können durch die Instruktion `new` angelegt werden.

new *Gefolgt von zwei Index-Bytes b1 und b2.*
Die beiden vorzeichenlosen Index-Bytes werden verwendet, um eine Position im Laufzeit-Konstanten-Pool der aktuellen Klasse zu berechnen: $pos = (b1 << 8) \mid b2$
An der berechneten Stelle im Konstanten-Pool muss eine Referenz zu einer Klasse stehen, von welcher eine Instanz im Heap angelegt wird. Die Referenz zum Objekt wird auf den Operand-Stack geschrieben.

instanceof *Gefolgt von zwei Index-Bytes b1 und b2.*
Die beiden vorzeichenlosen Index-Bytes werden verwendet, um eine Position im Laufzeit-Konstanten-Pool der aktuellen Klasse zu berechnen: $pos = (b1 << 8) \mid b2$
An der berechneten Stelle im Konstanten-Pool muss eine Referenz zu einer Klasse stehen. Der oberste Wert des Operand-Stacks, welcher eine Referenz zu einem Objekt im Heap sein muss, wird von dort entfernt. Ist das referenzierte Objekt eine Instanz der Klasse, die im Argument angegeben wurde, schreibt die Instruktion den *integer*-Wert 1 auf den Operand-Stack, sonst den Wert 0.
So kann geprüft werden, ob ein gegebenes Objekt die Instanz einer bestimmten Klasse ist.

Die Instanz einer Klasse kann persistente Speicher haben, die von allen Methoden und Threads ausgelesen und beschrieben werden können (je nach Zugriffsbeschränkung). Diese Felder werden durch die nachfolgenden Instruktionen angesprochen.

getfield *Gefolgt von zwei Index-Bytes b1 und b2.*
Die beiden vorzeichenlosen Index-Bytes werden verwendet, um eine Position im Laufzeit-Konstanten-Pool der aktuellen Klasse zu berechnen: $pos = (b1 << 8) \mid b2$
An der berechneten Stelle im Konstanten-Pool muss eine Referenz zum Feld einer Klasse stehen. Die Instruktion entfernt den obersten Wert vom Operand-Stack, welcher ein Verweis zu einem Objekt der Klasse sein muss, aus der das Feld ausgelesen werden soll. Der Wert des gefundenen Feldes wird auf den Operand-Stack geschrieben.

putfield Gefolgt von zwei Index-Bytes b1 und b2.
Die Klasse, in die ein Feld beschrieben werden soll, wird genau wie eben erläutert berechnet. Die Instruktion entfernt die obersten beiden Werte vom Operand-Stack. Der erste muss ein Verweis zu einem Objekt der Klasse sein, in die das Feld geschrieben werden soll. Der zweite ist der Wert, welcher in das Feld geschrieben werden soll.

getstatic Analog zu `getfield` mit dem Unterschied, dass keine Objektreferenz auf dem Operand-Stack liegen muss.

putstatic Analog zu `putfield` mit dem Unterschied, dass keine Objektreferenz auf dem Operand-Stack liegen muss.

Arrays

Arrays werden wie Objekte behandelt: Gearbeitet wird mit einer Referenz, die in einer Variablen abgelegt werden kann. Zum Lesen aus und zum Speichern in Arrays gibt es spezielle Bytecode-Instruktionen, die nachfolgend aufgezählt werden.

iaload *Keine Argumente.*
Entfernt die Objekt-Referenz ref und die Position $index$ vom Operand-Stack, wobei ref zu einem Array zeigen muss, dessen Elemente vom Typ *integer* sind, und lädt den Wert aus dem Array, welcher an der Position des *integer*-Wertes $index$ gespeichert ist, auf den Operand-Stack.

iastore *Keine Argumente.*
Entfernt die Objekt-Referenz ref, die Position $index$ und den Wert w vom Operand-Stack, wobei ref zu einem Array zeigen muss, dessen Elemente vom Typ *integer* sind, und speichert den Wert w an die Position $index$ im Array.

Analog hierzu existieren die Instruktionen

- baload und bastore (für den Datentyp *byte*),
- saload und sastore (für den Datentyp *short*),
- caload und castore (für den Datentyp *char*),
- laload und lastore (für den Datentyp *long*),
- daload und dastore (für den Datentyp *double*),
- faload und fastore (für den Datentyp *float*), sowie
- aaload und aastore (für Verweise auf Objekte als Element-Typ).

Erzeugt werden Arrays mit den Instruktionen `newarray`, `anewarray` und `multianewarray`.

newarray *Gefolgt von einem Byte $atyp$.*
Entfernt den obersten Wert anz vom Operand-Stack, welcher den Typ *integer* haben muss, erstellt ein Array der Länge anz und dem Datentyp, welcher mit $atyp$ referenziert wird und schreibt eine Referenz zum erstellten Array auf den Operand-Stack.
Das Argument $atyp$ kann folgende Werte haben: 4 (*boolean*), 5 (*char*), 6 (*float*), 7 (*double*), 8 (*byte*), 9 (*short*), 10 (*integer*) oder 11 (*long*).

anewarray *Gefolgt von zwei Index-Bytes $b1$ und $b2$.*
Entfernt den obersten Wert anz vom Operand-Stack, welcher den Typ *integer* haben muss, erstellt ein Array der Länger anz und schreibt eine Referenz zum erstellten Array auf den Operand-Stack. Das Index-Byte $b = (b1 << 8) \mid b2$ gibt eine Position im Laufzeit-Konstanten-Pool der aktuellen Klasse an. Das Element an dieser Stelle muss ein symbolischer Verweis zu einer Klasse sein, welche als Datentyp für die Elemente des Arrays verwendet wird.

multianewarray *Gefolgt von zwei Index-Bytes $b1$ und $b2$, sowie einem Byte dim.*
Analog zu `anewarray` mit den folgenden Unterschieden:
(1) Das Argument dim, welches als vorzeichenloser Wert gelesen wird, gibt die Anzahl der Dimensionen an, die das multidimensionale Array haben soll.
(2) Auf dem Operand-Stack müssen dim integer-Werte, welche die Größen der Dimensionen angeben.

A.1.3. Kontrollfluss

Einache Verzweigung

Für die unbedingte Verzweigung existiert die Bytecode-Instruktion `goto`.

goto *Gefolgt von zwei Verzweigungs-Bytes $b1$ und $b2$.*
Springt zur Stelle $pos + ((b1 << 8) + b2)$ im Programm, wobei pos die Position der Instruktion `goto` ist, gemessen vom Anfang der aktuellen Methode an, und setzt die Ausführung dort fort.

Zum Beenden einer Methode stehen die folgenden Instruktionen zur Verfügung:

return *Keine Argumente.*
Beendet die Ausführung der Methode und springt zum Aufrufer.

<t>return *Keine Argumente.*
Entfernt den obersten Wert vom Operand Stack und schreibt ihn auf den Operand-Stack des Aufrufers. Der Wert muss bei der Instruktion `ireturn` den Typ *integer* haben, bei `lreturn` den Typ *long*, bei `freturn` den Typ *float*, bei `dreturn` den Typ *double* sowie bei `areturn` den Typ *reference*.

athrow *Keine Argumente.*
Entfernt den Ersten Wert vom Operand-Stack, der ein Verweis zu einem Objekt sein muss, das von `Throwable` ableitet. Die Objektreferenz wird als Fehler geworfen.

Bedingte Verzweigung

Der Befehlssatz einer JVM beinhaltet weiterhin Anweisungen zum Vergleichen von Zahlen- und Referenzwerten. Beim Vergleich von *integer*-Werten wird die Ausführung bei einem positiven Ergebnis grundsätzlich an einer anderen Stelle im Programm fortgesetzt. So ist die Umsetzung von If-Anweisungen und Schleifen möglich.

if_icmp<bdg> *Gefolgt zwei Verzweigungs-Bytes* $b1$ *und* $b2$*.*
Entfernt die beiden obersten Werte vom Operand-Stack, die den Typ *integer* haben müssen. Ist der Vergleich von ihnen positiv, wird die Ausführung an der Stelle $pos + ((b1 << 8) + b2)$ fortgesetzt, wobei pos die Position der Instruktion `if_icmp<bdg>` ist, gemessen vom Anfang der aktuellen Methode an.
Der Ausgang des Vergleichs der Werte a und b fällt abhängig von der Bedingung positiv aus.
`if_icmpgt`: Positiv, falls $a > b$.
`if_icmplt`: Positiv, falls $a < b$.
`if_icmpge`: Positiv, falls $a \geq b$.
`if_icmple`: Positiv, falls $a \leq b$.
`if_icmpeq`: Positiv, falls $a = b$.
`if_icmpne`: Positiv, falls $a \neq b$.

Analog dazu ist der Vergleich von Verweisen möglich:

if_acmp<bdg> *Gefolgt von zwei Verzweigungs-Bytes* $b1$ *und* $b2$*.*
Analog zu `if_icmp<bdg>`, mit dem Unterschied, dass die beiden obersten Werte auf dem Operand-Stack vom Typ Verweis sein müssen.
Die möglichen Bedingungen sind hier:
`if_acmpeq`: Positiv, falls $a = b$.
`if_acmpne`: Positiv, falls $a \neq b$.

Für *integer*-Vergleiche mit Null existiert der folgende Typ an Instruktionen.

if<bdg> *Gefolgt von zwei Verzweigungs-Bytes* $b1$ *und* $b2$*.*
Entfernt den obersten Wert vom Operand-Stack, der den Typ *integer* haben muss und vergleicht ihn mit der Zahl Null. Fällt der Vergleich positiv aus, wird die Ausführung genau wie bei der Instruktion `if_icmp<bdg>` an einer anderen Stelle im Programm fortgesetzt.
Der Ausgang des Vergleichs des Werts mit 0 fällt abhängig von der Bedingung positiv aus.
`ifgt`: Positiv, falls $a > 0$.
`iflt`: Positiv, falls $a < 0$.
`ifge`: Positiv, falls $a \geq 0$.
`ifle`: Positiv, falls $a \leq 0$.
`ifeq`: Positiv, falls $a = 0$.
`ifne`: Positiv, falls $a \neq 0$.

Für Vergleiche von Werten, die den Typ *long*, *float* oder *double* haben, gibt es die nachstehenden Instruktionen.

lcmp *Keine Argumente.*
Entfernt die obersten beiden Werte a und b vom Operand-Stack, die den Typ *long* haben müssen und schreibt den *integer*-Wert 1 auf den Operand-Stack, falls $a > b$, 0, falls $a = b$ und -1, falls $a < b$.

fcmp<op> *Keine Argumente.*
Entfernt die obersten beiden Werte a und b vom Operand-Stack, die den Typ *float* haben müssen und schreibt den *integer*-Wert 1 auf den Operand-Stack, falls $a > b$, 0, falls $a = b$ und -1, falls $a < b$.
Für den Umgang mit nicht definierten Zahlen gibt es die beiden Formen `fcmpl` und `fcmpg`. Falls einer der beiden Werte NaN ist, schreibt `fcmpl` den Wert -1 und `fcmpg` den Wert 1 auf den Operand-Stack.

dcmp<op> Analog zu `fcmp<op>` für den Datentyp *double*.

Mehrfache Verzweigung

Mittels der **`switch`**-Anweisung ist es in Java-Programmen möglich, anhand einer Reihe an Bedingungen zu einer anderen Stelle im Code zu springen. Es gibt dabei die zwei Formen `tableswitch`, die besonders schnell ist, sowie `lookupswitch`, die einen größeren Bereich an Bedingungen abdecken kann.

tableswitch *Gefolgt von einer variablen Anzahl an Bytes.*
Die Argumente sind wie folgt aufgebaut:

- 0-3 Bytes Padding, sodass das *standardByte1* an einer durch 4 teilbaren Position steht, gemessen ab dem Beginn der aktuellen Methode
- *standardByte1*, *standardByte2*, *standardByte3*, *standardByte4*
- *startByte1*, *startByte2*, *startByte3*, *startByte4*
- *endeByte1*, *endeByte2*, *endeByte3*, *endeByte4*

Direkt nach dem Padding folgen drei 32-bit *integer*-Werte $standard$, $start$ und $ende$, dargestellt als Zweierkomplement. Daran schließen $ende - start + 1$ weitere 32-bit *integer*-Werte an, die Offsets. Alle Werte werden wie folgt berechnet:

$$wert = (byte1 << 24) \mid (byte2 << 16) \mid (byte3 << 8) \mid byte4$$

Die `tableswitch`-Instruktion entfernt den obersten Wert w vom Operand-Stack, der vom Typ *integer* sein muss und prüft, ob gilt $start \leq w \leq ende$. Ist dies nicht der Fall, wird die Ausführung an der Stelle $pos + standard$ fortgesetzt, wobei pos die Position der Instruktion `tableswitch` ist, gemessen vom Anfang der aktuellen Methode an.
Andernfalls wird der Offset $w - start$ zum Berechnen der Position zum Fortsetzen der Ausführung verwendet.

lookupswitch *Gefolgt von einer variablen Anzahl an Bytes.*
Die Argumente sind wie folgt aufgebaut:

- 0-3 Bytes Padding, sodass das *standardByte1* an einer durch 4 teilbaren Position steht, gemessen ab dem Beginn der aktuellen Methode
- *standardByte1*, *standardByte2*, *standardByte3*, *standardByte4*
- *npaare1*, *npaare2*, *npaare3*, *npaare4*

Direkt nach dem Padding folgend zwei 32-bit *integer*-Werte $standard$ und $npaare$, dargestellt als Zweierkomplement. Daran schließen $npaare$ Paare von weiteren 32-bit *integer*-Werten an. Der jeweils erste Wert ist der Schlüssel eines Paars, der zweite stellt den Offset dar.
Die `lookupswitch`-Instruktion entfernt den obersten Wert vom Operand-Stack und vergleicht diesen mit den Schlüsseln der Werte-Paare. Sobald einer der Werte passt, wird die Ausführung an der Stelle $pos + offset$ fortgesetzt, wobei pos die Position der `lookupswitch` ist, gemessen vom Anfang der aktuellen Methode an, sowie $offset$ der Offset des Werte-Paars. Passt keiner der Schlüssel zum Wert w, wird die Ausführung an der Stelle $pos + standard$ fortgesetzt.

Methodenaufrufe

Im Befehlssatz einer JVM sind auch Methodenaufrufe als Bytecode-Instruktionen verfügbar. Die unterschiedlichen Arten von Aufrufen sollen im Folgenden dargestellt werden.

invokestatic *Gefolgt von zwei Index-Bytes b1 und b2.*
Aufruf von als `statisch` deklarierten Methoden.
Die beiden vorzeichenlosen Index-Bytes werden verwendet, um eine Position im Laufzeit-Konstanten-Pool der aktuellen Klasse zu berechnen: $pos = (b1 << 8) \mid b2$
An der berechneten Stelle im Konstanten-Pool muss eine Referenz zu einer Methode stehen. Weiterhin müssen auf dem Operand-Stack die Argumente der aufgerufenen Methode liegen. Sie werden entfernt und gegebenenfalls durch das Ergebnis der Methodenausführung ersetzt.

invokevirtual
Aufruf von Methoden in anderen Klassen-Instanzen.
Analog zu `invokestatic` mit dem Unterschied, dass vor den Argumenten auf dem Operand-Stack die Referenz zu einer Klassen-Instanz liegen muss. Auf diesem Objekt wird der Methodenaufruf durchgeführt.

invokespecial
Aufruf von privaten Methoden, solchen zur Instanziierung und in Eltern-Klassen.
Analog zu `invokevirtual`.

invokeinterface *Gefolgt von zwei Index-Bytes b1 und b2, sowie einem Byte count und einem Byte mit dem Wert 0.*
Aufruf von Interface-Funktionen auf einem Objekt. Für mehr Informationen siehe [LY99].

invokedynamic *Gefolgt von zwei Index-Bytes b1 und b2, sowie zwei Bytes mit dem Wert 0.*
Dynamischer Methoden-Aufruf. Für mehr Informationen siehe [LY99].

A.1.4. Stack-Operationen

Durch die Instruktionen *pop* und *pop2* kann der oberste bzw. die beiden obersten Werte auf dem Operand-Stack gelöscht werden. Die Instruktion *dup* schreibt den ersten Wert des Operand-Stacks ein weiteres Mal an den Anfang. *dup_x1* führt diese Aktion mit dem zweiten Wert durch und *dup_x2* dupliziert den dritten Wert an den Anfang. Analog dazu arbeiten *dup2*, *dup2_x1* und *dup2_x2* und führen die Operationen mit jeweils zwei Bytes durch.

Mit der Anweisung *swap* ist es zudem möglich, die beiden obersten Werte zu vertauschen.

Literaturverzeichnis

[BDR14] F. Berg, F. Durr, K. Rothermel. Increasing the Efficiency and Responsiveness of Mobile Applications with Preemptable Code Offloading. In *Mobile Services (MS), 2014 IEEE International Conference on*, S. 76–83. IEEE, 2014. (Zitiert auf den Seiten 9 und 18)

[BLC02] E. Bruneton, R. Lenglet, T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002. (Zitiert auf Seite 19)

[CBC+10] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, S. 49–62. ACM, New York, NY, USA, 2010. doi:10.1145/1814433.1814441. URL http://doi.acm.org/10.1145/1814433.1814441. (Zitiert auf den Seiten 9, 18 und 19)

[CIM+11] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, S. 301–314. ACM, New York, NY, USA, 2011. doi:10.1145/1966445.1966473. URL http://doi.acm.org/10.1145/1966445.1966473. (Zitiert auf Seite 17)

[Cro06] D. Crockford. JSON: The fat-free alternative to XML. In *Proc. of XML*, Band 2006. 2006. (Zitiert auf Seite 30)

[FN01] L. M. Feeney, M. Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, Band 3, S. 1548–1557. IEEE, 2001. (Zitiert auf Seite 9)

[jik] Jikes Research Virtual Machine. URL www.jikesrvm.org. (Zitiert auf den Seiten 31 und 35)

[LY99] T. Lindholm, F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd Auflage, 1999. (Zitiert auf den Seiten 11, 45 und 53)

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. E. Lorensen, et al. *Object-oriented modeling and design*, Band 199. Prentice-hall Englewood Cliffs, 1991. (Zitiert auf Seite 11)

[SSX+12] A. Saarinen, M. Siekkinen, Y. Xiao, J. K. Nurminen, M. Kemppainen, P. Hui. Can Offloading Save Energy for Popular Apps? In *Proceedings of the Seventh ACM International Workshop on Mobility in the Evolving Internet Architecture*, MobiArch '12, S. 3–10. ACM, New York, NY, USA, 2012. doi:10.1145/2348676.2348680. URL http://doi.acm.org/10.1145/2348676.2348680. (Zitiert auf Seite 9)

Alle URLs wurden zuletzt am 10. 05. 2015 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift