Inhaltsverzeichnis

1 Einleitung

1.1 Einleitende Worte

Java hat sich in den letzten Jahren zu einer der wichtigsten Programmiersprachen entwickelt. Seine Plattformunabhängigkeit und seine Sicherheitskonzepte zeichnet sie gegenüber vielen anderen Programmiersprachen aus. Allerdings wurden diese Vorteile durch eine geringere Effizienz erkauft. Da Javaprogramme in Form eines Bytecodes verteilt werden, der dann von einer virtuellen Maschine interpretiert wird, blieben sie in der Performanz deutlich hinter C-Programmen zurück. Diese Situation besserte sich mit dem Aufkommen von JIT-Compilern, die den Bytecode vor dem Ausführen zuerst in Maschinencode übersetzten. Insbesondere Java-Programme, die viele kleine Methoden hatten, sind aber immer noch langsamer als äquivalente C++-Programme. In flexiblen und nach objektorientierten Kriterien entworfenen Programmen tauchen solche 'feingranularen Strukturen' leider sehr häufig auf.

Es gibt Konzepte für optimierende JIT-Compiler, die auch für solche Programme schnellen Code erzeugen können. Dabei wird intensiv das Einbinden von Methodenaufrufen eingesetzt.

1.2 Zielsetzung

In dieser Diplomarbeit soll ein JIT-Compiler für eine existierende JVM implementiert werden. Der Compiler soll in Java nach objektorientierten Kriterien entwickelt werden. Dabei sollte auf die Portierbarkeit geachtet werden. Die Optimierungsmöglichkeiten eines JIT-Compilers für Java sollten diskutiert werden.

1.3 Aufbau der Ausarbeitung

Diese Diplomarbeit läßt sich grob in 4 Teile aufteilen:

Der erste Teil besteht aus Kapitel 2 und 3. In Kapitel 2 wird die Java Virtual Machine besprochen. Dabei sollen insbesondere die Aspekte besprochen werden, die für einen JIT-Compiler relevant sind. In Kapitel 3 geht es um die JVM von MetaXa. Es wird auf die reflektive Programmierung eingegangen und die Besonderheiten dieser JVM besprochen.

Der zweite Teil besteht aus Kapitel 4 und 5. Kapitel 4 ist eine kurze Beschreibung des Zielprozessors Intel 386. In Kapitel 5 geht es um die Technik der Just-In-Time-Compilation. Hier werden die Anforderungen an JIT-Compiler behandelt. Dabei wird auch darauf eingegangen, welche Lösungen in vorhandenen JIT-Compilern existieren, um diese Anforderungen zu erfüllen. Im letzten Teil des Kapitels wird noch auf verschiedene existierende JIT-Compiler eingegangen.

Im dritten Teil, dem Kapitel 6 und 7, geht es um das Design und die Implementierung des JIT-Compilers für MetaXa. Zuerst werden dort die Ziele und Prioritäten für den Entwurf genannt. Darauf folgt eine Übersicht über den gesamten Compiler und die Beschreibung der verschiedenen Aspekte des Entwurfs.Im Kapitel 7 geht es um die Erweiterungen und Verbesserungen des Compilers. Diese wurden nur zum Teil implementiert.

Im vierten und letzten Teil besteht aus einer Beschreibung der Compilertests in Kapitel 8, und einer kurzen Bewertung des Entwurfs in Kapitel 9.

2 Die JVM

2.1 Konzept

Java ist eine objektorientierte Programmiersprache, die 1995 von der Computerfirma Sun eingeführt wurde. Sie wurde in den Jahren davor von James Gosling und anderen entwickelt [Arnold u.a. 96] . Programme, die in dieser Sprache geschrieben sind, haben gegenüber konventionellen Programmiersprachen folgende Vorteile:

- Sie sind auf unterschiedlichen Prozessoren und Betriebssystemen ohne Neucompilierung ausführbar (Portierbar)
- Fehlerhafte Programme können den Rechner nicht zum Absturz bringen (Robustheit)
- Es ist möglich, die Funktionen, die ein Programm ausführen darf, einzuschränken, so daß bösartige Programme keinen Schaden anrichten können.(Sicherheit)
- Eine zumindest im Vergleich zu anderen interpretierten Sprachen sehr gute Performanz.
- Teile des Programms werden erst geladen, wenn sie benötigt werden (Dynamisches Laden). Damit ist es zum Beispiel möglich, Programmcode auszuführen, der erst nach dem Start des Programms erzeugt wurde.

Diese Eigenschaft wurden realisiert, indem eine spezielle Ausführungsumgebung für Java Programme entwickelt wurde, die Java Virtual Machine (JVM). Eine JVM stellt einen vom Betriebssystem und Hardware unabhängige Ausführungsumgebung zur Verfügung. Die Komponenten dieser Ausführungsumgebung sollen in diesem Kapitel besprochen werden [Lindholm u.a. 96], [Meyer u.a. 97], [Dalheimer 97].

Um Javaprogramme auf einer JVM auszuführen, wird es von einem Java Compiler (z.B. javac von Sun) in eine Zwischensprache, den Java Bytecode übersetzt. Der Bytecode einer Klasse wird mit einigen zusätzlichen Informationen in einer sog. Klassendatei (Classfile) gespeichert. Diese Klassendateien können dann von einer JVM geladen und ausgeführt werden.

2.2 Anforderungen an eine Implementierung

Insbesondere um die gute Portierbarkeit und die Robustheit von Java Programmen gewährleisten zu können müssen an die Implementierung einer JVM sehr strenge und exakt definierte Anforderungen gestellt werden. [Lindholm u.a. 96]

Von diesen Anforderungen ist auch der JIT-Compiler betroffen, da sich auch der von ihm erzeugte Code an diese Regeln halten muß.

2.3 Aufbau der JVM

2.3.1 Die Ausführungseinheit

Die Ausführungseinheit (Execution Engine) ist das Herz einer jeden virtuellen Maschine, sozusagen ihr virtueller Prozessor. Sie liest den Bytecode der einzelnen Methoden und führt ihn aus. Um diese Aufgabe zu erfüllen kann sie auf alle anderen Komponenten der JVM zugreifen. Da fast alle Ereignisse in einer JVM durch das Ausführen von Bytecode ausgelöst werden, kontrolliert die Ausführungseinheit das Verhalten der gesamten JVM.

Normalerweise wird sie als Interpreter implementiert. Dieser liest der Reihe nach die Bytecodes, und führt je nach Bytecode verschiedene Aktionen aus (sog. action procedures).

Um eine Methode auszuführen erstellt der Interpreter für sie eine Laufzeitumgebung (Stackframe). Diese enthält den Speicher für die lokalen Variablen, die Operanden und einige weitere Daten.

Zum Allokieren des Speichers für die Laufzeitumgebung gibt es für jeden Thread des Javaprogramms einen eigenen Stack, den Javastack.

Obwohl das Konzept der JVM eher an einer Implementierung als Interpreter orientiert ist, kann man sie natürlich auch noch anders realisieren. Nur muß sich diese Implementierung nach dem Reference Manual richten, sich damit genauso wie ein Interpreter verhalten.

Die wichtigsten Alternativen für die Realisierung einer Ausführungseinheit sind die Implementierung in Hardware und die Implementierung mithilfe eines JIT-Compilers.

Eine Beschreibung des Befehlssatzes für diese Ausführungseinheit findet sich im Kapitel 2.3.4.

2.3.2 Die Methodendatenbank

Eine JVM muß natürlich einiges an Daten über das auszuführende Programm speichern. Dafür dient die Methodendatenbank (methodarea).

Diese speichert im wesentlichen die Datenstrukturen der Klassendateien, die um JVM-spezifische Informationen erweitert wurden. Abbildung 1 zeigt die wichtigsten Elemente einer Klassendatei.

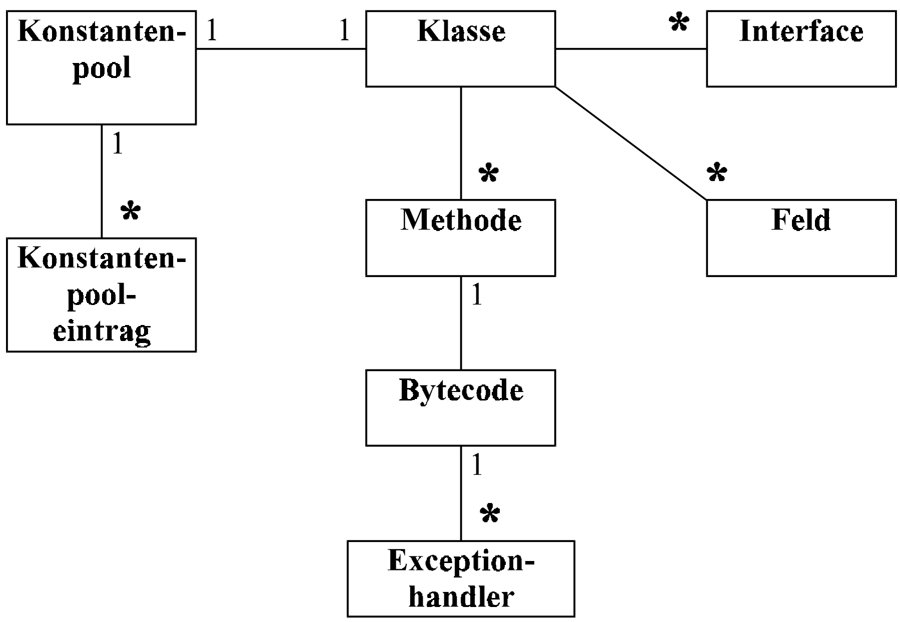

Abbildung 1: Die wichtigsten Elemente einer Klassendatei

Es ist sinnvoll, auch die statischen Daten des Programms in der Methodendatenbank zu speichern.

Eine Klasse in der Methodendatenbank durchläuft bis zur ersten Verwendung drei Stufen: Zuerst wird die Klasse geladen. Dabei werden die Datenstrukturen der Klasse meist von der Festplatte in die JVM gelesen. Anschließend wird die Klasse mit den Strukturen der Methodendatenbank verknüpft (linking). Erst danach ist es für ein Javaprogramm möglich, die Klasse zu verwenden. Zuletzt wird sie initialisiert. Dabei werden die Werte der statischen Felder gesetzt, und die Initialisierungsmethoden aufgerufen.

Wenn eine Klasse geladen wird, so dürfen auch alle von dieser Klasse referenzierten Klassen geladen und verknüpft werden. Nur mit der Initialisierung einer Klasse muß man solange warten, bis sie zum ersten Mal ‘aktiv’ verwendet wird. Aktiv verwendet wird eine Klasse, wenn zum Beispiel auf ein nichtkonstantes Feld zugegriffen wird, oder eine Methode der Klasse aufgerufen wird [Lindholm u.a. 96, Kapitel 2.16.3, Linking].

Für den JIT-Compiler ist das relevant, da er Code erzeugen muß, der bei der ersten aktiven Verwendung eine Initialisierung auslöst. Eine Initialisierung zur Compilezeit wäre nicht korrekt.

2.3.3 Der Konstantenpool

In einer Klassendatei sind die Konstanten der Klasse in einer einzigen geordneten Tabelle zusammengefaßt, dem Konstantenpool. Diese Konstantenpools werden auch als Datenstruktur in die Methodendatenbank übernommen. Im Bytecode und an anderen Stellen in der Klassendatei werden die Konstanten dann durch Indizes in den jeweiligen Konstantenpool referenziert. Dieses Kapitel beschreibt eine etwas vereinfachte Sicht des Konstantenpools.

Tabelle 1 zeigt eine Tabelle der wichtigen Konstantenpooleinträge. Diese lassen sich in zwei Gruppen aufteilen:

In die numerischen Konstanten und Stringkonstanten, sowie in die symbolischen Referenzen auf andere Klassen bzw. die Komponenten von anderen Klassen. Von Interesse ist insbesondere die zweite Gruppen von Einträgen.

Bevor ein Interpreter einen Konstantenpooleintrag verwenden kann muß dieser aufgelöst werden. Bei der Konstantenpoolauflösung wird dem symbolische Eintrag ein reales Objekt in der JVM zugeordnet. Wenn der Konstantenpooleintrag zum Beispiel eine Klasse referenziert, so muß bei der Auflösung man prüfen, ob die referenzierte Klasse bereits geladen ist, und sie gegebenenfalls laden und für die JVM vorbereiten. Dem Eintrag wird dann diese Klasse zugeordnet.

Die Konstantenpoolauflösung kann entweder für die gesamte Klasse stattfinden, nachdem diese geladen wurde (statische Konstantenpoolauflösung), oder für jeden Eintrag einzeln, wenn er von einem Bytecodebefehl verwendet wird (späte Konstantenpoolauflösung). [Lindholm u.a 96]

Typ	Beschreibung
Integer	Integerkonstante 32 Bit
Float	Fließkommakonstante 32 Bit
Long	Integerkonstante 64 Bit
Double	Fließkommakonstante 64 Bit
String	Stringkonstante
Class	Klassenreferenz (vollständiger Name der Klasse)
FieldRef	Referenz auf Feld (Name, Typ und Klasse des Feldes)
MethodRef	Referenz auf Methode (Name, Typ und Klasse der Methode)
InterfaceMethodRef	Referenz auf Interfacemethode (Name, Typ und Klasse)

Tabelle 1: Vom Bytecode referenzierte Konstantenpooleinträge

2.3.4 Der Java Bytecode

Der Java Bytecode ist die Sprache des virtuellen Prozessors der JVM. In den Klassendateien ist der Code der Javamethoden in diesem Format gespeichert. Die folgende Beschreibung des Bytecodes konzentriert sich ausschließlich auf die für den JIT-Compiler wichtigen Aspekte.

Um die Befehle des Java Bytecodes zu verstehen muß man zuerst den Aufbau der Laufzeitumgebung einer Methode, dem Stackframe, kennen. Aus konzeptueller Sicht besteht dieser aus zwei Bereichen, den lokalen Variablen und dem Operandenstack. Jeder dieser Bereiche ist in vier Byte große Zellen aufgeteilt. Die Anzahl dieser Zellen hängt von der jeweiligen Methode ab. Da der Operandenstack wie ein klassischer Stack verwendet wird, existiert für ihn ein Stackzeiger. Für die Implementierung der JVM muß die Laufzeitumgebung auch noch andere Daten erhalten. Diese sind aber für den Bytecode nicht sichtbar.

Der auffallendste Unterschied zwischen einem normalen Hardware-Prozessor und der dem virtuellen Prozessor der JVM ist, daß es sich bei letzterem um eine sogenannte Stackmaschine handelt.

Das heißt, die meisten Befehle holen ihre Operanden zuerst vom Operandenstack, führen damit eine Operation aus, und legen das Ergebnis zurück auf den Operandenstack. Durch dieses Vorgehen muß man für die Bytecodebefehle keine Operanden mitspeichern, da diese implizit durch den Stackzeiger des Operandenstacks gegeben sind. Die Bytecodebefehle können daher sehr kompakt gespeichert werden, Eine Ausnahme von diesem Schema bilden nur einige Kopierbefehle, die Daten zwischen den lokalen Variablen und dem Operandenstack kopieren, sowie ein einziger Befehl zum Addieren einer Konstante mit einer lokalen Variable.

Durch den stackorientierten Befehlssatz läßt sich ein Interpreter für Java einfacher und effizienter implementieren. Für einen JIT-Compiler stellt der Befehlssatz aber eine zusätzliche Schwierigkeit dar.

Um zum Beispiel den Ausdruck z = (x+1)*y auszuwerten, könnte man den folgenden Bytecode verwenden:

```
iload x     ; x auf den Stack legen
iconst_1    ; 1 auf den Stack legen
iadd        ; addieren
iload y     ; y auf den Stack legen
imul        ; multiplizieren
istore z    ; Ergebnis von Stack nehmen und in z schreiben
```

Die Buchstaben bezeichnen in diesem Beispiel lokale Variablen. In realem Bytecode würden an ihrer Stelle die Indizes von lokalen Variablen stehen.

Gespeichert werden die Befehle in einem einfachen Format: Jeder Befehl besteht aus einem Byte für den Befehlstyp (opcode) und keinem, einem oder mehreren ganzzahligen Konstanten als Parameter. Manchen Befehlen kann man ein Prefix-Byte voranstellen, das die Breite der Parameter modifiziert.

2.3.5 Der Heap

Auf dem Heap wird während der Programmausführung der Speicher für die Objekte allokiert. Dieser Speicher muß über eine automatische Freispeicherverwaltung (einem Garbage Collector) verfügen, da in Java nicht mehr benötigte Objekte nicht explizit freigegeben werden.

Verschiedene Implementierungen des Heaps unterscheiden sich in der Art des Zugriffs auf die Objekte, und in der Art der Freispeicherverwaltung. Diese Unterschiede müssen von einem JIT-Compiler berücksichtigt werden.

Es gibt die Möglichkeit, daß Referenzen auf Objekte als direkte Zeiger auf den Objektspeicher implementiert sind [Microsoft 98]. Alternativ dazu kann man eine Referenz als einen Zeiger auf ein Handle realisieren, das einen Zeiger auf den eigentlichen Objektspeicher enthält.

Mache Garbage Collektoren benötigen sogenannte Stackmaps. Aus diesen können sie erfahren, welche Speicherzellen des Stacks zu einem bestimmten Zeitpunkt Referenzen auf Objekte im Heap enthalten. Soll ein JIT-Compiler mit einem solchen Collektor zusammenarbeiten, so muß der Compiler in der Lage sein, Stackmaps für den compilierten Code zu erstellen.

2.3.6 Der Stack

Ein Interpreter allokiert die Laufzeitumgebungen für die Methoden in einem eigenen Speicherbereich, dem Javastack. Für jeden Thread des Javaprogramms existiert ein solcher Stack. Da diese im allgemeinen von JIT-compilierten Methoden nicht verwendet werden, soll er hier nicht weiter diskutiert werden.

2.3.7 Der Methodenaufruf bei Interpretern

Bei einem Methodenaufruf wird bei der JVM zuerst die Objektreferenz und anschließend die Argumente auf den Operandenstack gelegt. Die Argumente haben dabei dieselbe Reihenfolge wie in der Methodensignatur.

Beispiel: Aufruf von ‘int add(int a, long b)’;

```
aload this
iload a
lload b
invokevirtual add(IJ)I
```

Die aufgerufene Bytecodemethode erwartet die Objektreferenz (this) in der lokalen Variablen 0 und die Argumente der Methode in der darauffolgenden lokalen Variablen. Der Operandenstack enthält vor dem Aufruf also dieselben Daten wie die lokalen Variablen der aufgerufenen Methode.

Diese Aufrufkonventionen wurden von den Entwicklern der JVM so gewählt um eine effiziente Implementierung des Methodenaufrufs zu erlauben. Die Lage des Bereichs der lokalen Variablen der aufgerufenen Methode kann so gewählt werden, daß sie mit dem Operandenstack der aufrufenden Methode überlappt. Auf diese Weise kann man das unnötige Kopieren von Argumenten vermeiden. Die Übergabe des Rückgabewerts kann dann dadurch geschehen, daß die aufgerufene Methode ihn in die lokale Variable 0 schreibt.

Für den JIT-Compiler ist das relevant, da auch er Code für Methodenaufrufe erstellen muß. Außerdem läßt sich durch die Aufrufkonvention der JVM das Inlining leichter implementieren.

2.3.8 Ausnahmen

Ausnahmen können bei der JVM aus folgenden Gründen ausgelöst werden:

- der Bytecode löste explizit eine Ausnahme aus (athrow)
- ein Bytecodebefehl wurde mit illegalen Operanden ausgeführt (z.B. ArithmeticException, ClassCastException)

- die Ressourcen der JVM reichten nicht aus, um eine Operation auszuführen (z.B. StackOverflowError bei Methodenaufruf) bzw. eine Klassendatei war fehlerhaft (z.B. NoSuchMethodError)

Wenn eine Ausnahme aufgetreten ist, so muß sie von der JVM behandelt werden. Dabei wird zuerst überprüft, ob für diesen Typ von Ausnahme in der aktuellen Methode ein Handler existiert. Wenn ja, so wird dieser Handler aufgerufen. Ansonsten wird der oberste Stackframe vom Javastack genommen, und die Ausnahmebehandlung bei der aufrufenden Methode auf die gleiche Weise fortgesetzt.

Für jeden Handler werden in der Klassendatei die folgenden Daten gespeichert:

- die Bytecodeadresse des Handlers
- der Bytecodebereich, in dem Ausnahmen von diesem Handler behandelt werden (der von diesem Handler geschützt ist)
- die Klasse von Ausnahmen, die der Handler akzeptiert

Diese Ausnahmebehandlung ist auch bei einem JIT-Compiler ein wichtiges Thema.

2.4 JSR und RET

Die Bytecodebefehle JSR und RET machen die Implementierung eines JIT-Compilers etwas komplizierter.

Diese Befehle dienen dazu, Unterprogramme aufzurufen (JSR, Jump SubRoutine), und von einem Unterprogrammaufruf zurückzukehren (RET, RETurn). Anders als bei den Befehlen zum Methodenaufruf ist hier das Unterprogramm ein Teil des Bytecodes der aktuellen Methode. Der Javacompiler verwendet sie normalerweise nur für die Erzeugung des Codes für Javas Finally-Konstrukt.

Diese Semantik von JSR und RET ist für den JIT-Compiler wichtig. Erstens weil er diese Befehle übersetzen muß, zweitens weil der Operandenpositionen bestimmen muß, und drittens weil er eventuell eine Stackmap (siehe Kapitel 2.3.5) berechnen muß.

Der erste Punkt ist problemlos, und der dritte wird zum Beispiel in [Agesen u.a. 97] behandelt. Der zweite Punkt soll hier noch genauer besprochen werden.

Ein Unterprogrammaufruf läuft wie folgt ab:

- 'JSR' legt die Adresse des nächsten Bytecodebefehls auf den Operandenstack, und springt zu der Adresse, die als unmittelbarer Parameter des Bytecodes angegeben ist
- das Unterprogramm kopiert die Rücksprungadresse vom Operandenstack in eine lokale Variable (Befehl astore)
- wenn das Unterprogramm fertig ist, so kann sie mit dem Befehl RET die Programmausführung an der Rücksprungadresse zurückkehren. Die lokale Variable mit der Rücksprungadresse wird dem Bytecodebefehl als unmittelbarer Parameter übergeben

Ein paar Experimente haben gezeigt, daß sich die Unterprogrammaufrufe nicht unbedingt wie 'normale' Unterprogrammaufrufe verhalten müssen. Normalerweise sollte ein Unterprogrammaufruf nur zu seinem unmittelbaren Aufrufer zurückkehren dürfen. Mit den

JSR/RET-Befehlen kann man Bytecode schreiben, in dem Unterprogramme zu einem anderen Aufrufer zurückkehren. Der folgende Bytecode wird vom Bytecodeverifier von Java akzeptiert (java -verify):

```
.method public static jsrTest()V
.limit locals 2
.limit stack 2
jsr A
return
A:
astore 1
jsr B
B:
ret 1
.end method
```

Für das Bestimmen der Operandenpositionen muß der Compiler den Code in 'Ausführungsreihenfolge' durchgehen. Dafür muß er für jeden Befehl die möglichen Nachfolger bestimmen können. Das ist mit der hier beschriebenen Semantik für den Befehl RET relativ aufwendig. Da Methoden wie die oben beschriebene in realen Klassendateien praktisch nicht vorkommen wurde dieses Problem vorerst ignoriert.

2.5 Erzeugen des Bytecodes für eine JVM

Wegen den Sicherheitsaspekten unterliegt auch der Code in Klassendateien relativ strengen Anforderungen. Zum Beispiel darf der Bytecode der einen Klasse nicht auf die privaten Felder einer anderen Klasse zugreifen.

Ein Compiler, der Code für eine JVM erzeugt, muß das berücksichtigen. Deshalb hat ein solcher Compiler von vornherein wesentlich weniger Möglichkeiten der Optimierung wie ein herkömmlicher Compiler. Zum Beispiel ist das Inlining von Methoden anderer Klassen nicht möglich, wenn diese auf private Felder zugreifen.

Und auch wenn die Methode der anderen Klasse nur auf öffentliche Felder zugreift, so müssen nach dem Inlining bei einer Änderung des Quellcodes der zweiten Klassen beide Klassen neucompiliert werden. Dies muß vom Programmierer überwacht werden, da viele Javacompiler dies nicht automatisch tun.

Ein weiteres Problem für einen optimierenden Javacompiler ist die Tatsache, daß die optimale Wahl von Bytecodebefehlen für eine Methoden von der Ausführungseinheit der JVM abhängt. Diese ist dem Javacompiler im allgemeinen aber nicht bekannt.

Das sind Gründe dafür, das zumindest einige Optimierungen vom JIT-Compiler in der JVM und nicht von einem Javacompiler ausgeführt werden müssen.

2.6 Verifikation von Methoden

Eine JVM kann einen sog. Klassendateiverifizierer (Class File Verifier) einsetzen, um fehlerhafte Klassendateien zu erkennen.

Wenn ein Klassendateiverifzierer eingesetzt wird, so kann ein JIT-Compiler davon ausgehen, daß die Quelldaten, und insbesondere der Bytecode, bestimmte Eigenschaften erfüllt.

Normalerweise wird der Verifizierer nur für Klassendateien verwendet, die über das Netz geladen werden. Dateien, die von dem lokalen Rechner stammen, werden nicht überprüft.

Es kann also durchaus vorkommen, daß der Compiler fehlerhafte Quelldaten als Eingabe bekommt. Die Frage ist, wie er darauf reagieren soll. In meiner Implementierung wurde dieses Problem ignoriert. Das sollte auch so in Ordnung sein, da das Verhalten der JVM bei fehlerhaften Klassendateien undefiniert ist. Die JVM von Sun zum Beispiel kann in solchen Fällen mit einem ‘Segmentation Fault’ abbrechen.

Idealerweise sollte der Compiler bei Fehlern in den Quelldaten eine passende Ausnahme auslösen.

3 MetaXa

3.1 Konzept

MetaXa ist eine nahezu vollständige Implementierung einer JVM, die von Michael Golm im Rahmen einer Diplomarbeit entwickelt wurde. Das vorrangige Ziel bei der Entwicklung von MetaXa war es, eine JVM zu entwerfen, die reflektive Programme ausführen kann.

3.2 Metaprogrammierung und reflektive Programmierung

Dieses Kapitel soll nur die für den JIT-Compiler wichtigen Begriffe der Metaprogrammierung erläutern. Eine exaktere Beschreibung findet sich in [Golm 97].

Bei der Metaprogrammierung wird ein System betrachtet, das aus zwei Schichten besteht: Dem Basissystem und dem Metasystem.

Beim Basissystem handelt es sich um ein ganz normales Programm, das Berechnungen des Anwendungsbereichs ausführt (z.B. ein Adressenverwaltungsprogramm).

Das Metasystem dagegen kann das Verhalten und die Struktur des Basissystems beobachten und gegebenfalls modifizieren. Es kann zum Beispiel Nachrichten, die zwischen den Objekten des Basissystems ausgetauscht werden, abfangen, und mit einer beliebigen Aktion darauf reagieren.

Ein solches System ist ein Beispiel für ein sog. reflektives Programm, d.h. ein Programm das sein eigenes Verhalten beobachten und verändern kann.

Interessant ist die Metaprogrammierung für die Implementierung von Konzepten wie die Persistenz von Objekten, RMI, Replikation oder der Migration. In einem Metasystem können diese Konzepte in der Metaebene programmiert werden, ohne daß das Basissystem davon tangiert wird. Dies führt dazu, daß die Struktur des Basissystems einfacher und anwendungsnäher wird.

3.3 Reifikation

Den Prozeß, normalerweise nicht verfügbare Strukturen und nicht beeinflußbare Ereignisse einem Programm zugänglich zu machen, nennt man Reifikation (‘Verdinglichung’). Für die Metaprogrammierung ist die Verfügbarkeit von Reifikation Voraussetzung, da man sonst das Verhalten und die Struktur eines Programmes nicht beobachten kann.

3.4 Strukturelle Reifikation

Java verfügt von Hause aus über die Möglichkeit der sog. strukturellen Reifikation, bei der ein Programm Informationen über die Struktur seiner Klassen, Methoden und Felder erhalten kann. Die Reflection API stellt die Methoden dafür zur Verfügung.

MetaXa verfügt über eine eigene API zur strukturellen Reifikation. Beide API’s unterscheiden sich in einigen Punkten. Wichtig für den Compiler ist, daß die API von MetaXa es erlaubt, den Bytecode von Methoden zu lesen. Dieser wird vom Compiler als Quellcode benötigt. Auch die anderen Informationen über den zu compilierenden Code sind für den Compiler auf diesem Wege zugänglich. Allerdings fehlen Daten, die von der Implementierung der JVM abhängen, wie zum Beispiel die Adressen von statischen Feldern.

3.5 Reifikation des Verhaltens

Die Besonderheit der MetaXa JVM ist die Reifikation von Verhalten. Das ist das Verfügbarmachen von dynamischen Ereignissen, die während der Programmausführung auftreten.

Konkret kann MetaXa die folgenden Ereignisse reifizieren:

- Methodenaufrufe (sowohl eingehende als auch ausgebende)
- Zugriffe auf Variablen von Objekten
- Sperren von Objekten
- Laden von Klassen
- Erzeugen von neuen Objekten

Man kann dabei sowohl Ereignisse, die einzelne Objekte betreffen, reifizieren, als auch die Ereignisse aller Objekte einer Klasse.

3.6 Bezug der Metaprogrammierung zum JIT-Compiler

Für die Implementierung eines JIT Compilers ist die Metaprogrammierung bzw. die Architektur von MetaXa aus den folgenden zwei Gründen von Bedeutung:

Erstens kann man die Möglichkeiten der Metaprogrammierung für die Implementierung ausnutzen. Zweitens muß man berücksichtigen, daß zu compilierende Methoden von der Reifikation betroffen sein können.

3.7 Der Compiler als Metaprogramm

Ein in Java geschriebener JIT-Compiler kann als Metaprogramm betrachtet werden, da er Teile des auszuführenden Javaprogramms als Eingabe verarbeitet und verändert. Im folgenden wird beschrieben, wie der Compiler die Möglichkeiten der Reifikation für sich nutzen kann.

Wie schon oben angedeutet, kann man das Interface für die strukturelle Reifikation dafür verwenden, um den Quellcode für die Compilierung zu erhalten.

Die Installation von compilierten Code läuft dagegen in der aktuellen Version noch nicht über das Interface für die Reifikation. Grundsätzlich ist aber auch das denkbar.

Ein JIT-Compiler muß dann ausgelöst werden, wenn eine Methode das erste Mal aufgerufen wird. Man kann daher eine Metaklasse implementieren, die die Methodenaufrufe einer Basisklasse überwacht, und beim jeweils ersten Aufruf einer Methoden den Compiler startet [Golm 97]. Der Nachteil dieser Lösung ist der relativ große Overhead.

3.8 Die Compilation mit reifizierter Klassen

Der Compiler muß berücksichtigen, daß mache Ereignisse, die den erzeugten Code betreffen, möglicherweise reifiziert sind bzw. noch reifiziert werden. Der compilierte Code müßte das natürlich berücksichtigen, und beim Auftreten eines entsprechenden Ereignisses die Kontrolle an die JVM übergeben, damit diese das Metaobjekt aufrufen kann.

Die einfachste Lösung für dieses Problem ist es, alle Methoden, die von der Reifikation betroffen sind, von der Compilation auszuschließen, bzw. compilierten Code zu invalidieren, wenn die Reifikation nach der Compilation passiert.

Da die Reifikation sowieso nur bei einem kleinen Teil der Methoden verwendet wird, und ohnehin zu einer schlechteren Performanz führt, wurde bei der Implementierung diese Lösung gewählt. Dadurch kann die Schnittstelle zwischen JVM und Compiler einfach gehalten werden.

Zuletzt kann man sich überlegen, ob es sich lohnt, reifizierte Methoden nicht nur zu compilieren, sondern auch just-in-time zu optimieren. Insbesondere kleine und einfache Meta-Methoden wie zum Beispiel die Implementierung eines Aufrufzählers könnte davon profitieren. Das erfordert dann aber einer Erweiterung der Schnittstellen der JVM.

4 Der Prozessor Intel 386

Der von mir implementierte Compiler soll ausschließlich Code für den Intel 386 und seine Nachfolgemodelle erzeugen. Daher wird in diesem Kapitel kurz auf das Programmiermodell für den Intel 386 eingegangen, soweit das für den Compiler relevant ist. Eine Beschreibung des Prozessors findet sich in seinem Reference Manual [Intel 86].

4.1 Der Befehlssatz

Beim Intel 386 handelt es sich um einen 32 Bit CISC Prozessor. Er verfügt über acht 32 Bit breite Register, die für allgemeine Zwecke zur Verfügung stehen. Von diesen wird normalerweise eines für den Stackzeiger und eines als Zeiger für die Laufzeitumgebung verwendet.

Die meisten Befehle des Prozessors haben zwei Operanden. Von diesen kann einer eine Hauptspeicheradresse oder eine Konstante sein, der andere muß ein Register sein. Bei manchen Befehlen müssen die Operanden in bestimmten, festen Registern sein. So erwartet zum Beispiel der Divisionsbefehl den Dividenden in den Registern EAX und EDX.

Wenn ein Befehl auf eine Hauptspeicheradresse zugreift, so wird diese im allgemeinsten Fall nach der folgenden Formel berechnet:

Adresse = BasisRegister + IndexRegister*Skalierungsfaktor + Offset

Der Skalierungsfaktor kann dabei die Werte 1,2,4 und 8 annehmen. Jeder der Summanden kann weggelassen werden.

Beispiele:

```
ADD EAX, [EDI+10]     ; addiere die Speicherzelle mit Adresse
                      ; EDI+10 zum Register EAX
INC EBX               ; erhöhe EBX um eins
SUB [EAX + EBX*4 + 16], ECX  ; subtrahiere ECX von der
                             ; Hauptspeicheradresse
                             ; EAX + EBX*4 + 16
AND EAX, 255          ; Und-Verknüpfung von EAX mit 255
MOV EAX, EBX          ; kopiere EBX nach EAX
SHL EAX, ECX          ; Linksshift von EAX um ECX Bits (andere
                      ; Register als ECX sind hier als zweiter
                      ; Operand nicht erlaubt).
IDIV EBX              ; teile die 64-Bit-Zahl aus EDX und EAX
                      ; durch EBX (der Dividend ist implizit)
```

4.2 Die Fließkommabefehle

Für Fließkommaberechnungen ist bei der Intel 386-Architektur der Prozessor Intel 387 verantwortlich. Dieser verfügt über acht 80 Bit breite Fließkommaregister. Diese Register werden, anders als bei den Integerregistern, wie ein Stack verwendet. Das bedeutet, daß die meisten Befehle implizit als ersten Operanden das oberste Stackelement nehmen. Der zweite Operand kann im Hauptspeicher stehen, oder ein beliebiges Fließkommaregister sein. Dieses wird dann allerdings relativ zum obersten Stackelement adressiert. Der Stack wächst nach unten.

Beispiele für Befehle:

```
FLD [EAX + 10]     ; erniedrige den Stackpointer, und
                   ; lade die Fließkommazahl aus der
                   ; Hauptspeicheradresse [EAX + 10] in das
                   ; oberste (bisher freie) Register
FADD ST0, ST3      ; addiere das Register ST3 zum Register ST0
FSIN               ; berechne den Sinus der Stackspitze (STO),
                   ; und speichere das Ergebnis in ST0 (der
                   ; Stackpointer wird nicht verändert)
FSTP [EAX +10]     ; kopiere das oberste Register in den
                   ; Hauptspeicher, und nimm das Register vom
                   ; Stack
```

Für einen Algorithmus zur Registerallokation stellt die Organisation der Register als Stack eine zusätzliche Schwierigkeit dar. In diesem Compiler wurde daher auf die Verwendung des gesamten Stacks verzichtet. Nur das oberste Register wird benutzt.

Der 387 verfügt über ein Kontrollwort, mit dem man die Genauigkeit der Berechnungen und die Art der Rundung programmieren kann. Außerdem kann man festlegen, welche Ereignisse ein Hardwaresignal auslösen sollen. Das ist für einen JIT-Compiler für Java interessant, da man dadurch den Prozessor so einstellen kann, daß er sich entsprechend dem JVM Referenzhandbuch verhält. Das bedeutet konkret, daß er sich an den Standard IEEE 754 halten muß, und daß er keine Ausnahmen auslösen darf.

4.3 Das Zeitverhalten der Befehle

Für die Generierung von effizienten Code ist es wichtig, das Zeitverhalten der Befehle zu kennen, so daß man für eine bestimmte Operation eine möglichst schnelle Maschinenbefehlsfolge findet.

Bei modernen Prozessoren, wie dem Intel Pentium, hängt die Zeit, die sie für die Ausführung einer Befehlsfolge brauchen, nicht nur von der Art der Befehle ab, sondern auch von der Reihenfolge der Befehle.

Wichtig sind hier vor allem die folgenden Regeln [Abrash 94]:

Wenn ein Befehl ein Register modifiziert, so sollte dieses Register innerhalb der nächsten Taktzyklen nicht zum Adressieren einer Speicherzelle verwendet werden. Ansonsten kann die Befehlsfolge aufgrund eines sogenannten ‘Address Generation Interlock’ (AGI) einen Taktzyklus zusätzlich brauchen. Wenn man in einer JVM auf ein Objekt über ein Handle zugreift, so können solche AGI’s relativ häufig auftreten.

Durch seine superskalare Architektur kann der Pentium manche Befehlspaare parallel in einem Taktzyklus ausführen. Dann dürfen die Befehle aber nicht voneinander abhängen. Codegeneratoren, die mit einem virtuellen Stack arbeiten, erzeugen aber häufig Befehlsfolgen, in denen das nicht der Fall ist.

Obwohl sich bei modernen Prozessoren, wie dem Intel Pentium, deutliche Verbesserungen der Ausführungszeiten durch geschickte Befehlsauswahl und Befehlsanordnung (instruction scheduling) erreichen lassen, wurde dieser Aspekt bei der Implementierung weitgehend vernachlässigt, um sie zu vereinfachen und die Compilezeit kurz zu halten.

Mache Eigenschaften moderner Prozessoren vereinfachen die effiziente Codeerzeugung auch. Die Tatsache, daß beim Intel Pentium bedingte Sprünge gleich viel Zeit brauchen, egal ob sie genommen werden oder nicht, ist ein Beispiel dafür.

Die neuesten Prozessoren, wie der Pentium Pro oder der Pentium II sind auch in der Lage während der Ausführung die Maschinenbefehle neu zu ordnen (Out-of-order execution). Es ist durchaus möglich, daß sich auf solchen Prozessoren der Geschwindigkeitsgewinn durch Instruction Scheduling relativiert.

5 JIT-Compiler

Diese Kapitel gibt einen Überblick über JIT-Compiler und beschreibt auftretende Probleme und Lösungen in existierenden Systemen.

5.1 Konzept

Just-In-Time-Compiler (JIT-Compiler) dienen als Interpreterersatz.

Statt den Code einer Methode zu interpretieren wird dieser beim ersten Aufruf der Methode in Maschinencode übersetzt, Anschließend und bei den folgenden Aufrufen wird dann nur noch der Maschinencode aufgerufen.

Auf diese Weise kann man die Geschwindigkeit der Ausführungseinheit deutlich erhöhen, ohne die Vorteile eines Interpreters aufzugeben. Insbesondere bleibt die Portabilität erhalten, da die Programme wie beim Interpreter in einer maschinenunabhängigen Form verteilt werden.

Dadurch, daß jede Methode erst dann compiliert wird, wenn sie aufgerufen wurde, werden Wartezeiten beim Start des Programms vermieden. Außerdem verschwendet man keine Zeit für Methoden die nie aufgerufen werden.

5.2 Aspekte bei der Implementierung

Im folgenden werden die Anforderungen und Probleme bei der Implementierung eines JIT-Compilers aufgezählt. Dabei wird auch auf die Lösungen für die Probleme in existierenden Compilern eingegangen.

Der Schwerpunkt dieses Kapitels liegt auf JIT-Compilern für Java. Allerdings stammen einige Konzepte auch aus JIT-Compilern für andere Sprachen.

5.2.1 Konservative Compilierung

Das JVM Referenzhandbuch [Lindholm u.a. 97] stellt an eine Implementierung einer JVM relativ strenge Anforderungen, um die Sicherheit, Robustheit und Portierbarkeit von Java Programmen zu gewährleisten. Diese Anforderungen muß natürlich auch ein JIT-Compiler erfüllen. Für den JIT-Compiler bedeutet das, daß er im Code erzeugen muß, der sich genauso wie interpretierter Code verhält.

Einen Compiler, der Programme so compiliert daß sie sich exakt nach der Sprachspezifikation verhalten, nennt man konservativ. Das scheint auf den ersten Blick selbstverständlich. Allerdings darf ein konservativer Compiler auch nicht das Verhalten von fehlerhaften Programmen verändern. Diese Eigenschaft haben zum Beispiel die meisten C-Compiler nicht. Denn ein optimiertes C-Programm verhält sich bei Programmfehlern z.B. oft anders als ein nicht optimiertes.

Diese Forderung der konservativen Compilierung verhindert oft eine effizientere Codegenerierung oder macht die Implementierung des Compilers wesentlich aufwendiger.

Die konservative Compilierung spielt insbesondere bei den folgenden Punkten eine Rolle:

- Auflösung des Konstantenpools
- Fließkommaberechnungen
- Ausnahmenbehandlung

Dies Punkte werden in den folgenden Kapiteln noch genauer behandelt.

5.2.2 Compilationsgeschwindigkeit

Da JIT-Compiler zur Ausführungszeit arbeiten ist eine hohe Compilationsgeschwindigkeit sehr wichtig. Sie sollte zumindest so groß sein, daß sich der Aufwand für das Compilieren einer Methoden in der späteren Ausführung amortisiert. In interaktiven Anwendungen wirkt ein langsamer JIT-Compiler besonders störend, da er zu schlechten Antwortzeiten führt.

Es ist nicht besonders schwer, einen extrem schnellen JIT-Compiler zu schreiben, wenn der erzeugte Code langsam sein darf. Denn man kann einen JIT-Compiler als einfachen Makroexpander implementieren. Dieser ersetzt einfach jeden Bytecodebefehl durch einen festen Block Maschinencode. [Deutsch 84]

Da ein so erzeugter Code ineffizient ist verwenden die meisten JIT-Compiler einen aufwendigeren Codegenerator.

5.2.3 Registerallokation

Die meisten Maschinenbefehle erwarten zumindest teilweise ihre Operanden in den Registern des Prozessors. Ein JIT-Compiler, der alle Variablen (d.h. lokale Variablen und Operanden) im Hauptspeicher hält, muß für jeden Bytecodebefehl erst einige der Variablen in Register laden, den Befehl ausführen, und dann das Register mit dem Ergebnis wieder zurückschreiben. Ein so erzeugter Code enthält viele überflüssige Kopieroperationen zwischen Registern und Hauptspeicher.

Das Ziel der Registerallokation ist es, solche Kopieroperationen zu vermeiden, indem man Variablen über einen längeren Zeitraum in einem Register hält. Da die Zahl der Prozessorregister normalerweise beschränkt ist, kann man nicht immer alle Variablen in Registern halten. Statt dessen muß man den Variablen die Register so zuordnen, daß die Zahl der Kopieroperationen minimiert wird.

Da die Compilation schnell sein soll können aufwendige Algorithmen zur Registerallokation hier nicht eingesetzt werden. Statt dessen behilft man sich bei JIT-Compilern mit Verfahren die auf einfachen Heuristiken basieren.

Bei Prozessoren mit vielen Registern, wie dem SPARC Prozessor oder dem Alpha 21264 kann man den Variablen während der gesamten Methode feste Register zuordnen [Cramer u.a. 97][Grafl 97]. Da Java-Methoden meist klein sind reicht die Zahl der Register oft aus, um alle Variablen in Registern zu halten. Sollte das nicht möglich sein, so muß man die überzähligen Variablen im Hauptspeicher halten.

Bei Prozessoren mit wenigen Registern, wie dem Intel 386, erscheint das nicht sinnvoll. Da einige Register für bestimmte Befehle benötigt werden, kann man nur drei, maximal vier Register frei verwenden. Daher ist es bei solchen Prozessoren besser, a priori alle Variablen im Hauptspeicher zu halten, und sie nur innerhalb eines elementaren Blocks in Register auszulagern [Wilkinson u.a. 97]. Wenn eine Variable im Hauptspeicher in einem Maschinenbefehl verwendet wird, wird ihr ein Register zugeordnet. Diese Zuordnung bleibt dann solange erhalten, bis das Register für einen anderen Zweck benötigt wird oder der elementare Block zu Ende ist.

Wenn ein neues Register benötigt wird, obwohl alle Register schon belegt sind, so benötigt man ein Verfahren, das eines der belegten Register auswählt und in den Hauptspeicher zurückschreibt. Dafür kann man im Prinzip alle Freigabealgorithmen aus der Literatur über die virtuelle Speicherverwaltung verwenden, da in beiden Fällen dasselbe Problem gelöst werden muß. In Kaffe [Wilkinson u.a. 97] wird ein Least-Recently-Used-Verfahren benutzt.

5.2.4 Die Konstantenpoolauflösung

Die Auflösung der Konstantenpooleinträge kann zur Compilezeit stattfinden, da sie nach dem JVM Reference Manual während der Verknüpfungsphase einer Klasse stattfinden darf (statische Auflösung) [Lindholm u.a. 96]. Die Konstantenpoolauflösung stellt für den Compiler damit kein Problem dar.

Etwas anderes ist die Initialisierung von Klassen. Diese darf erst stattfinden, wenn die Klasse das erste Mal aktiv verwendet wird. Das kann bei der ersten Objekterzeugung, bei einem Methodenaufruf und bei einem statischen Feldzugriff der Fall sein. Da Objektallokationen sowieso von der JVM übernommen werden, stellen sie kein Problem für den Compiler dar. Auch Methodenaufrufe werden zuerst als Aufrufe an die JVM übersetzt.

Problematisch sind nur die statischen Feldzugriffe. Hier muß beim ersten Zugriff die JVM aufgerufen werden, um die Initialisierung durchzuführen. Bei den folgenden Aufrufen ist nur noch das Lesen bzw. Schreiben einer Datenzelle notwendig. Cacao [Grafl 97] ignoriert das Problem: Hier findet bei statischen Feldzugriffen die Initialisierung zur Compilezeit statt. Der JIT-Compiler von Sun [Cramer u.a. 97] verwendet selbstmodifizierenden Code um das Problem zu lösen: Der Bytecodebefehl wird zuerst als Aufruf an die JVM übersetzt, die die Initialisierung durchführt. Nach der Initialisierung wird der Aufruf aus dem Code entfernt.

5.2.5 Retargetability

Ein großer Vorteil von Java ist seine gute Portabilität: Programme können ohne Neucompilation auf einem neuen Rechner ausgeführt werden. Voraussetzung dafür ist allerdings, daß für diesen Rechner eine JVM existiert. Daher ist eine gute Portabilität für eine JVM eine wichtige Eigenschaft.

Für eine JVM mit JIT-Compiler reicht es nicht aus, daß der Programmcode der JVM auf dem neuen System compiliert werden kann. Denn auch der JIT-Compiler muß so angepaßt werden, daß er Maschinencode für das den neuen Rechner erzeugt. Einen Compiler, der Code für verschiedene Rechner erzeugen kann, nennt man im Englischen 'retargetable'.

Bei den konventionellen Compilern gibt des das Konzept der sog. Register-Transfer-Languages, mit dem sich optimierende Compiler implementieren lassen, die allein durch das Neuschreiben einer Maschinenbeschreibung Code für neue Prozessoren erzeugen können. Allerdings ist dieses Konzept für JIT-Compiler zu langsam. [Davidson u.a. 84b]

Daher verwendet man bei JIT-Compilern ein einfacheres und älteres Prinzip: Man teilt den JIT-Compiler in zwei Module auf, dem Frontend und dem Backend. Ersteres enthält den maschinenunabhängigen Code, Letzteres den Code der vom jeweiligen Zielsystem abhängt. Um den JIT-Compiler zu portieren muß nur das Backend neugeschrieben werden.

Entscheidend ist dabei die Sprache in der die Programmdaten vom Frontend an das Backend übergeben werden. Dabei sollte es sich um eine Sprache handeln, die möglichst einfach in Maschinensprache übersetzt werden kann, und die es dem Backend dennoch ermöglicht, effizienten Code zu erzeugen. Natürlich ist diese Zwischensprache unabhängig vom Zielrechner.

Der JIT-Compiler von Kaffe [Wilkinson u.a. 97] wurde nach diesem Prinzip entworfen. Als Zwischensprache wurde eine Dreiadreßsprache verwendet, die dem Bytecode von Java relativ ähnlich ist. Das Backend für die verschiedenen Prozessoren ist dabei sehr kompakt gehalten. Für jeden Befehl der Zwischensprache hat man eine Funktion die die Registerallokation durchführt und unmittelbar die Binärdarstellung erzeugt. Für die Registerallokation wird dabei ein Modul verwendet, das unabhängig vom Zielsystem ist. Der JIT-Compiler von Kaffe wurde auf die Prozessoren Intel 386, SPARC und Alpha portiert.

5.2.6 Optimierung

Da JIT-Compiler schnell sein müssen wenden sie normalerweise nur einfache Optimierungsverfahren an. Beispiele für solche einfachen Verfahren sind die Registerallokation, die verzögerte Codeerzeugung oder die Beseitigung redundanter Bereichsüberprüfungen.

Allerdings können auch aufwendigere Optimierungsverfahren für JIT-Compiler sinnvoll sein. Man muß sich dann aber auf die Teile des Programms beschränken, die häufig ausgeführt werden. Denn nur so kann man damit rechnen, daß sich der zusätzliche Aufwand für die Compilation amortisiert.

Dabei muß berücksichtigt werden, das ein JIT-Compiler natürlich nur den von ihm erzeugten Code optimieren kann. Wenn sich eine Ausführungseinheit nur 40 % der Zeit in compiliertem Code verbringt, so kann ein optimierender Compiler keine große Verbesserung bringen. Die Prozentzahl von 40 % ist dabei durchaus ein typischer Wert für manche Java-Programme. Die restlichen 60 % werden dabei für die Freispeicherverwaltung, die Synchronisation und für native Methoden benötigt. Eine Verbesserung des Codegenerators sollte daher mit einer allgemeinen Optimierung der Ausführungseinheit einhergehen.

Im folgenden werden einige Optimierungsverfahren beschrieben, die in JIT-Compilern verwendet wurden.

5.2.6.1 Auswahl optimaler Adressierungsmodi

Ein Compiler kann die Bytecodesequenz

```
iload 1
iconst 42
iadd
istore 1
```

mit dem Maschinenbefehl (Intel 386)

```
add EAX, 42
```

übersetzen. Bei diesem Beispiel wurde davon ausgegangen, daß sich die lokale Variable 1 im Register EAX befindet.

Am einfachste läßt sich ein solches Ergebnis mit einem sogenannten virtuellen Stack erreichen. [Cramer u.a. 97][Pittman u.a. 92]. Man geht dabei den Quellcode zum Übersetzen Befehl für Befehl durch. Dabei werden aber manche Befehle nicht sofort übersetzt, sondern nur ihre Wirkung gespeichert. Erst wenn man weiß, in welchen Kontext die Ergebnisse der verzögerten Befehle verwendet werden, wird tatsächlich Code erzeugt. Dadurch können oft Quellcodebefehle zusammengefaßt und besserer Maschinencode erzeugt werden. Die Technik ist auch unter dem Namen verzögerte Codeerzeugung bekannt.

Die Datenstruktur, die zum Speichern der Wirkung nicht sofort ausgeführter Befehle verwendet wird ist ein Array das genauso viele Elemente hat wie der Stackframe Variablen.

Daher der Name virtueller Stack. Jedes Element des Arrays speichert die Befehle, die ausgeführt werden müssen, um den Wert der entsprechenden Variable zu erhalten.

Für JIT-Compiler wird normalerweise eine einfachere Variante des virtuellen Stacks verwendet. Dabei werden nur Kopierbefehle und das Laden von Konstanten verzögert. Alle anderen Befehle werden sofort übersetzt. Auf diese Weise wird zwar nicht die Anzahl der arithmetischen Operationen reduziert, aber die Operanden dieser Befehle werden wesentlich besser gewählt.

5.2.6.2 Inlining

Inlining ist eine der wichtigsten Optimierungstechniken für Compiler von objektorientierten Sprachen. Da in objektorientierten Programmen oft viele kleine Methoden auftreten, lassen sich dadurch gute Verbesserungen erreichen.

Die Technik des Inlining ist einfach: Statt eine Methode aufzurufen wird der Rumpf der Methode in die aufrufende Methode eingesetzt. Dabei müssen natürlich die lokalen Variablen entsprechend angepaßt werden.

Aufgrund der Einschränkungen durch den Bytecode Verifier kann ein Java-nach-Bytecode Compiler das Inlining nur sehr begrenzt einsetzen (siehe dazu Kapitel 2.5).

In der Literatur finden sich keine Beispiele von JIT-Compilern für Java, die das Inlining beherrschen. Sun hat 1997 angekündigt, einen solchen Compiler zu entwickeln. Seitdem gibt es darüber aber keine weiteren Veröffentlichungen. Es ist aber durchaus wahrscheinlich, daß bei dem einen oder anderen kommerziellen JIT-Compiler einfaches Inlining implementiert wurde. Erwähnt wurde das Inlining von Java Methoden in [Cramer u.a. 97] und [Griswold 97]. Diese Artikel beschreiben aber keine Implementierung.

In seiner Dissertation beschreibt Craig Chambers die Implementierung einer Ausführungsumgebung für die objektorientierte Sprache Self [Chambers 92]. Dabei wird intensiv das Inlining eingesetzt, um ein Programm während der Ausführung zu optimieren. Allerdings ist Self eine Sprache bei deren Entwurf mehr Wert auf theoretische Eleganz als auf Effizienz gelegt wurde. Mit den folgenden Optimierungstechniken kann man bei Self daher größere Geschwindigkeitsgewinne erzielen als bei Java, das schon von Hause aus effizienter ist. Die Ergebnisse lassen sich dennoch teilweise auf einen Java JIT-Compiler übertragen. Die für diese Zwecke relevanten Aspekte der Arbeit sollen daher im folgenden beschrieben werden.

Die Schwierigkeit beim Inlining in objektorientierten Sprachen ist, daß ein virtueller Methodenaufruf, eine Nachricht, mehrere mögliche Empfänger haben kann. Das liegt daran, daß eine Objektreferenz zu verschiedenen Zeitpunkten verschiedene dynamische Typen haben kann, also auf Objekte verschiedenen Typs zeigen kann. Bei Inlining muß die aufgerufene Methode aber zur Compilezeit bekannt sein. Damit scheint das Inlining von virtuellen Methodenaufrufen auf den ersten Blick völlig unmöglich zu sein.

Er gibt grundsätzlich zwei Möglichkeiten, um Aufrufe virtueller Methoden doch in die aufrufende Methode einzubinden:

Erstens kann man mittels Datenflußanalyse versuchen, die möglichen Typen des Empfängers einzuschränken. Wenn sich dabei herausstellt, daß als Empfänger eines Aufrufs nur ein Typ in Frage kommt, so ist ein Inlining möglich. In [Chambers 92] ist diese Technik unter dem Namen ‘Typenanalyse’ beschrieben.

Die andere Möglichkeit ist die, den Code einer Methode so umzuformen, daß die Empfänger der darin auftretenden Aufrufe bekannt sind. Davon gibt es wiederum drei Varianten.

Bei der Customization wird für jeden möglichen Typ des impliziten Parameters (this) einer Methode eine eigene Version der Methode erzeugt. In jeder einzelnen dieser Methoden ist dann der dynamische Typ von 'this' bekannt. Dieselbe Technik funktioniert auch mit den expliziten Parametern einer Methode, die Implementierung wird dann aber wesentlich aufwendiger.

Die anderer Technik wird als 'Type Casing' (Fallunterscheidung nach Typen) bezeichnet. Dabei wird ein virtueller Aufruf durch eine Fallunterscheidung nach den Empfängertyp ersetzt. In jedem Pfad der Fallunterscheidung ist dann der dynamische Typ bekannt, und man kann das Inlining einsetzen. Hat man zum Beispiel die Klasse Polygon mit den Unterklassen Dreieck und Viereck, und der Methode draw(), so kann man den Aufruf

```
aPolygon.draw();
```

durch folgenden Code ersetzen:

```
if (aPolygon instanceof Dreieck)
    ((Dreieck)aPolygon).draw();
else if (aPolygon instanceof Viereck)
    ((Viereck)aPolygon).draw();
```

Die Fallunterscheidung bedeutet natürlich zusätzlichen Aufwand, und es ist nicht selbstverständlich, daß diese Umformung überhaupt einen Gewinn bringt. Allerdings kann ein nachgeschalteter Optimierungsalgorithmus mit dem umgeformten Code wahrscheinlich mehr anfangen.

Die dritte Möglichkeit für Codeumformungen, die das Inlining möglich machen, ist das Splitting. Da diese für einen JIT-Compiler für Java wenig Sinn macht will ich sie hier nicht beschreiben. Kurz gesagt wird beim Splitting das Verschmelzen der Ausführungspfade nach einer If-Then-Bedingung verzögert. Dadurch verfügt der nachfolgenden Code eventuell über detailliertere Typeninformation.

Es ist sinnvoll, bei diesen Umformungstechniken die Häufigkeitsverteilung der Typen zu berücksichtigen. Eine kurze Diskussion des Inlinings unter der JVM findet sich im Kapitel 7.1.

5.2.6.3 Partielle Auswertung

Bei der partiellen Auswertung versucht man, Anweisungen des Programms schon zur Compilezeit auszuwerten, und nicht erst zur Laufzeit. Möglich ist das natürlich nur, wenn die Operanden der Anweisung Konstanten sind. Partielle Auswertung kann weitgehend schon vom Java-nach-Bytecode Compiler durchgeführt werden. Daher lohnt sie sich bei einem JIT-Compiler nur nach dem Inlining. Diese Optimierungstechnik läßt sich sehr effizient implementieren.

Die partielle Auswertung wird in der Literatur über JIT-Compiler für Java nicht erwähnt. Nur der Compiler von Chambers [Chambers 92] verwendet diese Technik.

5.2.7 Behandlung von Ausnahmen

Wenn eine Ausnahme (exception) auftritt, so muß man die Methode in der Aufrufhierarchie finden, die diese Ausnahme behandelt, und die Ausführung beim der Ausnahmebehandlungsroutine (exception handler) dieser Methode fortsetzen. Dabei muß der Stack bereinigt (stack unwinding) werden. Das heißt, die nun nicht mehr benötigten Stackframes müssen entfernt werden.

Die Implementierung von Ausnahmen ist bei JIT-Compilern etwas komplizierter als bei Interpretern. Denn bei JIT-Compilern muß der Maschinenstack bereinigt werden. Wenn interpretierter Code zusammen mit compiliertem Code ausgeführt wird, so muß man den Javastack und den Maschinenstack parallel bereinigen.

Dafür gibt es bei den existierenden JIT-Compilern zwei unterschiedliche Lösungen:

Bei der ersten Lösung übernimmt die JVM die gesamte Ausnahmebehandlung. Um das Auftreten einer Ausnahme zu signalisieren ruft der compilierte Code oder der Interpreter eine bestimmte Routine der JVM auf. Die JVM sucht dann einen Handler für die Exception und entrollt dabei beide Stacks. Erst wenn sie einen Handler gefunden hat, übergibt sie die Kontrolle wieder an den Interpreter bzw. den compilierten Code. Diese Lösung ist zum Beispiel im JIT-Compiler von Sun [Cramer u.a. 97] oder in Kaffe [Wilkinson u.a. 97] implementiert.

Bei dieser Lösung muß die JVM das Layout der Stackframes auf dem Maschinenstack kennen. Dieses ist natürlich vom jeweiligen Betriebssystem und Prozessor abhängig. Dafür ist bei dieser Variante der JIT-Compiler etwas einfacher, da er kaum Code für die Ausnahmebehandlung erzeugen muß. Er muß der JVM nur mitteilen, welche Bereiche des compilierten Codes von welchem Handler geschützt werden, und wo die compilierten Exceptionhandler beginnen.

Bei der zweiten Lösung wird die Ausnahmebehandlung für den compilierte Code vollständig vom compilierten Code selbst übernommen. Das Werfen einer Ausnahme wird dabei mit einem Code übersetzt, der zuerst testet, ob die Ausnahme in der aktuellen Methode einen Handler hat. Wenn dem so ist, so wird unmittelbar dieser Handler angesprungen. Ansonsten wird die Methode beendet und zur aufrufenden Methode zurückgekehrt. Dabei wird ihr mitgeteilt, daß eine Ausnahme aufgetreten ist. Die aufrufenden Methode wird dann diese Ausnahme erneut werfen. Diese Art der Ausnahmebehandlung wurde in Cacao [Grafl 97] implementiert.

Das Problem dabei ist, daß man der aufrufenden Methode mitteilen muß, ob eine Ausnahme aufgetreten ist. In Cacao wird dafür neben dem Rückgabewert eines Methodenaufrufs ein Zeiger in einem Register an die aufrufenden Methode übergeben. Dieser ist NULL, wenn keine Ausnahme aufgetreten ist, und zeigt ansonsten auf das geworfene Ausnahmeobjekt. Nach jedem Aufruf wird überprüft, ob dieser Zeiger gesetzt ist.

Diese Lösung führt zu einem geringfügig höheren Aufwand bei Methodenaufrufen, und zu einer Vergrößerung des compilierten Codes.

Ein weiterer, davon unabhängiger Aspekt bei der Implementierung der Ausnahmen sind die Tests auf Referenzen, die Null sind. Das JVM Reference Manual verlangt, daß wenn über eine Referenz, die Null ist, auf ein Feld zugegriffen wird bzw. eine virtuelle Methode aufgerufen wird, eine Ausnahme ausgelöst wird. Die konservative Lösung für dieses Problem ist, vor jedem Feldzugriff und Methodenaufruf die Referenz zu testen und gegebenfalls eine Ausnahme auszulösen.

Bei Prozessoren, die beim Zugriff auf einen Nullzeiger ein synchrones Hardwaresignal auslösen, ist aber eine schnellere Lösung möglich. Man installiert einen Signalbehandler für diese Ausnahme (einem Segmentation-Fault), der dann die Java-Ausnahmebehandlung startet. Das wurde bei Kaffe [Wilkinson u.a. 97] so gemacht. Dasselbe Problem tritt bei Tests auf Divisionen von Integern und Longs durch Null auf, und es wird in Kaffe auch genauso gelöst.

5.2.8 Fließkommaberechnungen

Fließkommaberechnungen müssen nach dem JVM Referenzhandbuch entsprechend dem Standard IEEE 754 durchgeführt werden. Für einen JIT-Compiler, der Code für den Intel 386 erzeugt, stellt das kein allzu großes Problem dar, da der mathematische Coprozessor Intel 387 diesem Standard entspricht.

Man muß allerdings dafür sorgen, daß keine Ausnahmen bei Überlauf, Unterlauf, Division durch Null und bei denormalisierten Operanden ausgelöst werden. Die Rundung muß so eingestellt sein, daß in Richtung Null gerundet wird. Beides kann man durch Setzen eines Kontrollwortes erreichen.

Außerdem muß man darauf achten, daß Berechnungen nicht mit zu großer Präzision durchgeführt werden (sic). Den eine JVM darf Berechnungen mit Float's nur mit 32 Bit Genauigkeit durchführen, und Berechnungen mit Double's nur mit 64 Bit. Man kann dieses Problem dadurch lösen, daß man die Zwischenergebnisse von Fließkommaberechnungen immer in der Speicher zurückschreibt. Dadurch werden sie korrekt gerundet. Ein Beispiel für dieses Vorgehen findet sich in Kaffe [Wilkinson u.a. 97].

Allerdings kann man effizienteren Code erzeugen, wenn man die Zwischenergebnisse im Stack des Intel 387 beläßt. In diesem Fall muß man die gewünschte Genauigkeit über das Kontrollwort des Prozessors einstellen. Ein Mischen von Berechnungen mit 32 und 64 Bit Genauigkeit sollte dann aber relativ schwer sein. Der JIT-Compiler von Borland [Crelier 97] nutzt den Stack des Intel 387 um effizienteren Code zu erzeugen.

5.2.9 Binärcodeerzeugung

Die eigentliche Erzeugung der Binärdarstellung der Maschinenbefehle kann sehr einfach implementiert werden. Kaffe verwendet für jeden Befehl der Zwischensprachen eine Routine, die die Binärdarstellung unmittelbar in den Speicherbereich für die compilierte Methode schreibt. Nur die Wert für die verwendeten Register und konstanten Operanden müssen noch nachträglich eingefügt werden. Beim Schreiben des Codes für einen solchen Codegenerator kann man natürlich einen Assembler zu Hilfe nehmen. Trotzdem ist diese Lösung relativ aufwendig, und der Code ist schwer lesbar.

5.3 Existierende Systeme

In diesem Abschnitt sollen Beispiele für existierende just-in-time Compiler beschrieben werden. Bestimmte Aspekte dieser Compiler sind schon im Kapitel 5.2 behandelt. Auf diese Aspekte wird hier nicht mehr eingegangen.

5.3.1 Sun

Der JIT-Compiler von Sun wurde für Intel- und SPARC-Prozessoren entwickelt. Er wird zum Beispiel für das Java Development Kit von Sun verwendet. Eine Beschreibung des Compilers findet sich in [Cramer u.a. 97].

Der Compiler verwendet keine Zwischendarstellung des Codes der Methode. An Optimierungen beherrscht er die Registerallokation, verfügt über einen virtuellen Stack, und er hat einen einfachen Algorithmus zum Beseitigen von Bereichstests bei Arrayzugriffen. Es kann interpretierten und compilierten Code mischen. Die Übergänge zwischen beiden Varianten sind mit Stubs implementiert. Die Ausnahmebehandlung erfolgt vollständig durch die JVM und nicht durch den compilierten Code. Der Heap verwendet eine konservative Freispeicherverwaltung, der Zugriff auf Objekte erfolgt über Handles.

5.3.2 Kaffe

Kaffe ist eine virtuelle Maschine zum Ausführen von Java Bytecode. Sie verfügt sowohl über einen Interpreter als auch über eine JIT-Compiler. Sie steht unter der GNU General Public Licence. Das heißt insbesondere, daß der Quellcode in C frei verfügbar ist. Leider existiert kaum Dokumentation darüber. Um Informationen über das Design zu bekommen, muß man daher den Quellcode untersuchen. Eine wesentliche Eigenschaft von Kaffe ist seine gute Portierbarkeit (Kapitel 5.2.5).

Bei der Übersetzung verwendet Kaffe drei Schichten:

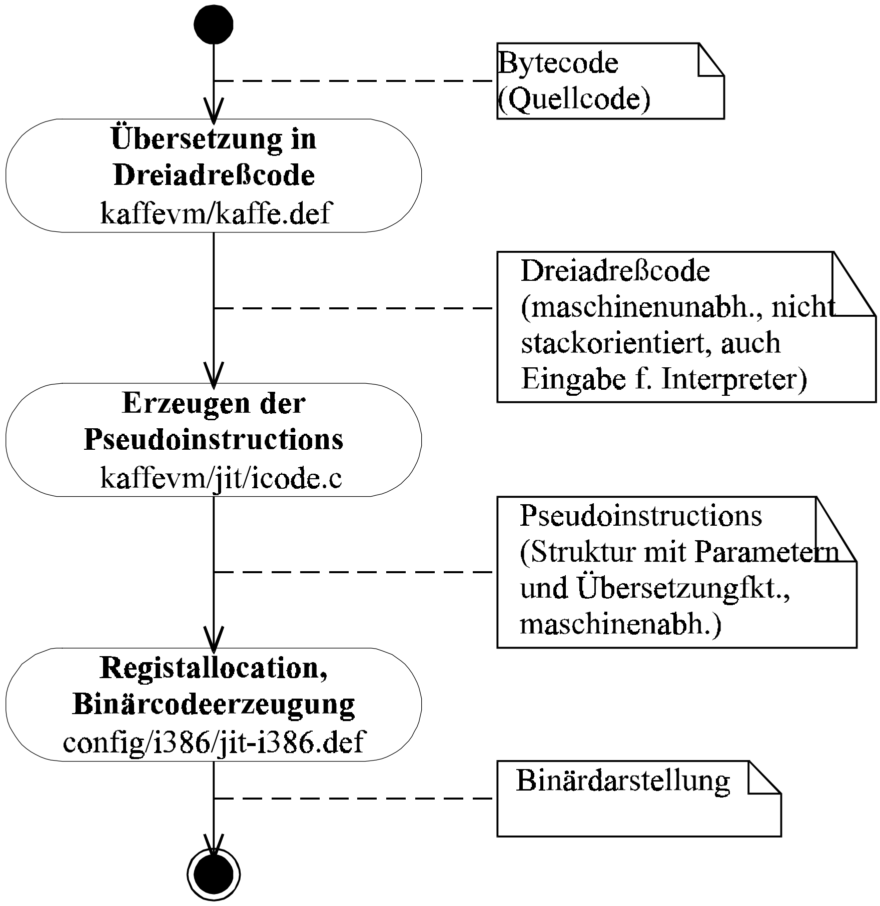

Abbildung 2: Schichten des Kaffe-JIT-Compilers

Der JIT-Compiler von Kaffe verwendet zwar einen Registerallokator, optimiert aber ansonsten relativ wenig.

5.3.3 Microsoft

Da es sich um ein kommerzielles Produkt handelt, ist über die Interna nur wenig bekannt. Die folgenden Informationen stammen im wesentlichen von der Website von Microsoft [Microsoft 98]. Der hier beschriebene JIT-Compiler wird zum Beispiel in Produkten wie dem Microsoft Explorer verwendet.

Eine wichtige Eigenschaft, die ihn von anderen JIT-Compilern unterscheidet, ist das Fehlen von Handles für den Zugriff auf Objekte über Referenzen.

Bei den meisten JIT-Compilern werden Objektreferenzen nicht als direkte Zeiger auf die Daten des Objekts implementiert. Statt dessen wird auf die Daten und Methoden von Objekten über Handles zugegriffen. Diese Art der Implementierung von Objektreferenzen vereinfacht die Freispeicherverwaltung. Sie führt aber auch zu einem langsameren Zugriff auf die Objektdaten, und benötigt zusätzlichen Verwaltungsaufwand für die Handles.

Eine weitere Eigenschaft der virtuellen Maschine von Microsoft ist daß sie einen nichtkonservativen Garbage-Collektor verwendet. Dies erfordert, daß die JVM zu jedem Zeitpunkt, an dem eine Freispeicherbereinigung ausgeführt werden kann, in der Lage sein muß die Lage aller Objektreferenzen aufzuzählen.

Da die JVM die Lage der Objektreferenzen bei JIT-compiliertem Code nicht kennt, muß der JIT-Compiler diese Informationen liefern. Zusätzlich zum compilierten Code muß der JIT-Compiler daher immer noch sogenannte Stackmaps berechnen, die die Lage der Referenzen in den Stackframes des compilierten Codes zu bestimmten Zeitpunkten beschreiben.

Die Entscheidung, einen nichtkonservativen Garbage-Collektor zu verwenden ist eine Konsequenz aus der Art und Weise wie die Objektreferenzen implementiert wurden. Denn ein konservativer Garbage-Collektor ist nur mit Handles einfach zu implementieren.

5.3.4 Borland

Auch der JIT-Compiler von Borland ist ein kommerzielles Produkt, dessen Interna nicht vollständig veröffentlicht werden. Allerdings existiert ein Interview mit dem Entwickler des Compilers, Regis Crelier, in dem er einige Entwurfsaspekte erläutert. [Crelier 97]

Der Compiler ist in C implementiert. Er verwendet zwei Optimierungsverfahren, die Registerallokation und den virtuellen Stack. Die Registerallokation ist zum Teil statisch und zum Teil dynamisch. Denn die ersten lokalen Variablen werden während der gesamten Methode festen Registern zugeordnet. Die übrigen Variablen werden dynamisch zugeordnet. Im Gegensatz zu Kaffe verwendet dieser JIT-Compiler den gesamten Stack des Fließkommaprozessors, wodurch er eine besonders gute Performanz bei Fließkommaberechnungen erreicht.

Die JVM kann interpretierten und compilierten Code gemeinsam ausführen. Für Aufruf von interpretierten Methoden werden Stubs verwendet, wodurch der Typ der aufgerufenen Methode für die compilierte Methode transparent ist. Auch die Auslösung des Compilers beim ersten Aufruf einer Methode erfolgt über Stubs.

5.3.5 Cacao

Cacao ist eine JVM mit JIT-Compiler die von Reinard Grafl im Rahmen einer Diplomarbeit am Institut für Computersprachen der Technischen Universität Wien entwickelt wurde. Der JIT-Compiler ist für einen 64 Bit RISC Prozessor, dem Dec Alpha, geschrieben. Der Compiler und die Laufzeitumgebung wurden in C implementiert [Grafl 97] .

Die Schwierigkeit beim Entwurf von Cacao war, daß die Zielmaschine 64 Bit breite Register hat, und das sie für Integer- und Fließkommaberechnungen unterschiedliche Registersätze verwendet. Sie unterscheidet sich damit deutlich vom Konzept der JVM, die mit 32 Bit breiten Worten arbeitet, und die Fließkommazahlen und Integers in gemeinsamen Variablen speichert. Man muß daher den Quellcode erst an die Zielmaschine anpassen. Eine direkte Übersetzung würde sehr ineffizienten Code erzeugen.

Die Zielmaschine hat jeweils 32 Fließkommaregister und 32 Integerregister. Das sind mehr als eine typische Methode für die Ausführung benötigt.

Die Registerallokation für lokale Variablen ist bei Cacao statisch. Allerdings kann man nicht jeder lokalen Variable ein einziges festes Register zuordnen, da Fließkommazahlen, Integerzahlen und Referenzen, bzw. 32- und 64-Bit Worte unterschiedlich behandelt werden müssen. Eine Variable kann aber verschiedene Datentypen enthalten. Cacao untersucht daher, welche Datentypen in welchen Variablen gespeichert werden, und allokiert für jeden Datentyp, der in einer Variable gespeichert wird, ein eigenes Register.

Die Register für temporäre Werte auf dem Operandenstack werden dagegen in der Reihenfolge des Auftretens allokiert.

Die Übersetzung selbst läuft wie folgt ab:

Zuerst wird der Code in sogenannte Superblöcke zerlegt. Das sind Codebereiche die jeweils von zwei Sprungziele begrenzt sind. Da diese Superblöcke größer sind als die elementaren Blöcke (auch Grundblöcke genannt), erlauben sie einem Compiler mehr Optimierungsmöglichkeiten.

Anschließend wird der Bytecode in einen Dreiadreßcode (Quadrupelcode) übersetzt. Dabei wird jeder Superblock für sich behandelt. In diesem Schritt erfolgt auch die Abbildung der Variablen auf Pseudoregister. Danach folgt die Beseitigung unnötiger Kopieroperationen (Copypropagation) und die Abbildung der Pseudoregister auf Maschinenregister. Anschließend wird der Maschinencode erzeugt. Im letzten Schritt werden die Sprungadressen eingetragen, da diese erst bekannt sind, wenn der Maschinencode erzeugt ist.

Das Auslösen der Compilation erfolgt bei Cacao über Stubs. Die Ausnahmebehandlung von Cacao wurde schon in Kapitel 5.2.7 beschrieben.

5.3.6 Smalltalk-80 von Schiffman/Deutsch

Bei der Smalltalk-80-Implementierung von Schiffman und Deutsch handelt es sich um das erste System, in dem ein JIT-Compiler eingesetzt wurde, um eine virtuelle Maschine zu optimieren. [Deutsch 84]

Einige Techniken, die in heutigen JIT-Compiler für Java verwendet werden wurden in dieser Arbeit das erste mal eingesetzt. Ein Beispiel dafür ist die Verwendung von selbstmodifizierenden Code um das Verhalten eines Interpreters effizient nachzuahmen.

Allerdings unterscheiden sich die Anforderungen an einen JIT-Compiler für Smalltalk und einen für Java in vielen Punkten. Daher wird auf eine detailliertere Beschreibung dieses Systems in dieser Arbeit verzichtet.

5.3.7 Self

Self ist eine objektorientierte Programmiersprache, die von David Ungar und Randy Smith als eine Verbesserung von Smalltalk entwickelt wurde. Es handelt sich dabei um eine rein objektorientierte Sprache, d.h. alle Daten sind durch Objekte repräsentiert, und die Interaktion zwischen Objekten erfolgt ausschließlich über Nachrichtenaustausch.

Beim Entwurf von Self wurde besonders auf die Eleganz, Flexibilität, Robustheit und Einfachheit der Sprache Wert gelegt, die Effizienz war zweitrangig.

Eine Folge dieser Entwurfsentscheidungen war, daß eine naive Implementierung eines Self-Interpreters extrem ineffizient ist.

Mit der Implementierung eines optimierenden JIT-Compilers zeigte Craig Chambers in seiner Dissertation [Chambers 92], daß es möglich ist, selbst für eine derartig aufwendige Sprache einen Compiler zu entwickeln, der eine Effizienz erreicht, die mit konventionellen Programmiersprachen vergleichbar ist. Sein Compiler erreichte Ausführungsgeschwindigkeiten, die nur um den Faktor zwei bis drei hinter optimierten C-Programmen zurück blieben.

Neben konventionellen Compilertechniken, wie globaler Registerallokation und CSE verwendet der Compiler auch Optimierungsverfahren, die speziell für objektorientierte Systeme bzw. speziell für Self entworfen wurden:

Die in diesem Kontext relevanten Verfahren wurden schon im Kapitel 5.2.6.2 beschrieben.

6 Design und Implementierung

6.1 Grundlegende Designentscheidungen

6.1.1 Ziele des Entwurfs

Das ursprüngliche Ziel des Entwurfs war es, einen JIT-Compiler zu schreiben, mit dem man verschiedene Optimierungstechniken und andere Konzepte ausprobieren kann. Änderungen und Erweiterungen sollten daher mit minimalen Aufwand zu realisieren sein. Durch einen klaren, robusten Aufbau und guten Debugging-Möglichkeiten sollten Fehler verhindert oder zumindest schnell gefunden werden können. Das Programm sollte weitgehend nach objektorientierten Kriterien entworfen sein.

Die Geschwindigkeit sollte ausreichend für einen JIT-Compiler sein, zumindest wenn der Compiler von einem optimierenden Compiler compiliert wird.

Der Compiler sollte grundsätzlich auch an andere JVM's anpaßbar sein.

6.1.2 Ausführungsgeschwindigkeit gegen Flexibilität

Wie in Kapitel 5.2.2 beschrieben ist eine hohe Compiliergeschwindigkeit eine wichtige Eigenschaft für eine JIT-Compiler. Das ist insbesondere für eine Implementierung in Java von Bedeutung, da Javaprogramme typischerweise deutlich langsamer ausgeführt werden als z.B. C-Programme.

Das gilt auch, wenn man die Möglichkeit berücksichtigt, daß ein JIT-Compiler, der in Java geschrieben ist, sich selbst compilieren kann. Wenn der Compiler also schnellen Code erzeugt, wird damit auch der Compiler selbst schnell ausgeführt werden. Allerdings erzeugen JIT-Compiler normalerweise langsameren Code als C-Compiler, teilweise weil sie schneller compilieren müssen, teilweise weil sie die hohen Anforderungen der JVM erfüllen müssen.

Aus diesem Grund scheint es sinnvoll zu sein, die Ausführungsgeschwindigkeit des Compilers zum primären Entwurfskriterium zu machen. Sowohl Interpreter als auch JIT-Compiler können Javaprogramme dann besonders schnell ausführen, wenn sie wenige Objekte allokieren, und Methodenaufrufe vergleichsweise selten vorkommen.

Will man allerdings ein flexibles und auf objektorientierten Kriterien basierendes Programm, so gelangt man oft zu einem Entwurf, der viele kleine Klassen mit relativ kurzen Methoden enthält. Entsprechend häufig sind dann Objekterzeugungen und virtuelle Methodenaufrufe.

Damit hat man zwei sich widersprechende Anforderungen, die hohe Ausführungsgeschwindigkeit und eine gute Flexibilität.

In meinem Entwurf habe ich nun den Schwerpunkt auf eine gute Flexibilität gelegt. Das Programm enthält daher relative viele kleine Methoden, und allokiert viele kleine Objekte.

Damit besteht die Gefahr, daß die Implementierung für eine JIT-Compiler zu langsam wird. Es ist bis jetzt nicht klar, ob sich durch nachträgliche Optimierungen die Geschwindigkeit ausreichend steigern läßt.

Normale JIT-Compiler können etwa 50000 Bytecodebefehle pro Sekunde übersetzen (100 MHz Pentium). Für jeden Befehl stehen also 20 us zur Verfügung.

Das reicht bei einem Interpreter für circa

- 3 Objekterzeugungen, oder für
- 25 Methodenaufrufe oder für
- 65 einfachere Bytecodebefehle (iadd ...)

aus.

Wenn man das Programm mit einem JIT-Compiler ausführt, sollte sich die Geschwindigkeit der Methodenaufrufe und Bytecodebefehle erhöhen (Faktor ca. 5), die der Objekterzeugung aber gleich bleiben.

Der aktuelle Entwurf benötigt dagegen deutlich mehr.

6.2 Einbettung in die Laufzeitumgebung

Abbildung 3 verdeutlicht die Einbettung des Compilers in die Umgebung.

Man sieht an diesem Diagramm, daß es zwei voneinander getrennte Ausführungsumgebungen gibt: Die native Umgebung und Ausführungsumgebung von Java. In der ersten wird die JVM und der compilierte Code des JIT-Compilers ausgeführt. In der zweiten läuft der Compiler und das Anwendungsprogramm.

Die beiden Umgebungen sind relativ stark voneinander isoliert. Ein Java Programm kann auf die native Umgebung nur über native Javamethoden zugreifen. Diese muß die JVM definieren.

Auf der anderen Seite kann die JVM nur sehr geringen Einfluß auf das auszuführende Javaprogramm nehmen. Denn sie muß sich an die Spezifikation für virtuelle Maschinen für Java halten. Ein Javaprogramm muß sie daher so ausführen, wie es durch den Programmcode festgelegt ist.

Die JVM kann allerdings selbständig Javaprogramme in isolierten Threads starten, um bestimmte Aufgaben zu erledigen.

Es gibt zwei Möglichkeiten, wie der Compiler gestartet werden kann:

Entweder direkt durch die JVM, oder durch eine Metaklasse. In beiden Fällen geschieht das über die Aufrufschnittstelle des Compilers.

Die wichtigsten Schnittstellen in diesem Entwurf sind die Compilezeit-Schnittstelle und die Laufzeit-Schnittstelle. Über erstere kann der Compiler die Quelldaten über das Programm erhalten. Über die letztere kann der compilierte Code mit der JVM kommunizieren.

Eine grobe Übersicht über die bei der Compilation auftretenden Aktivitäten gibt Abbildung 4.

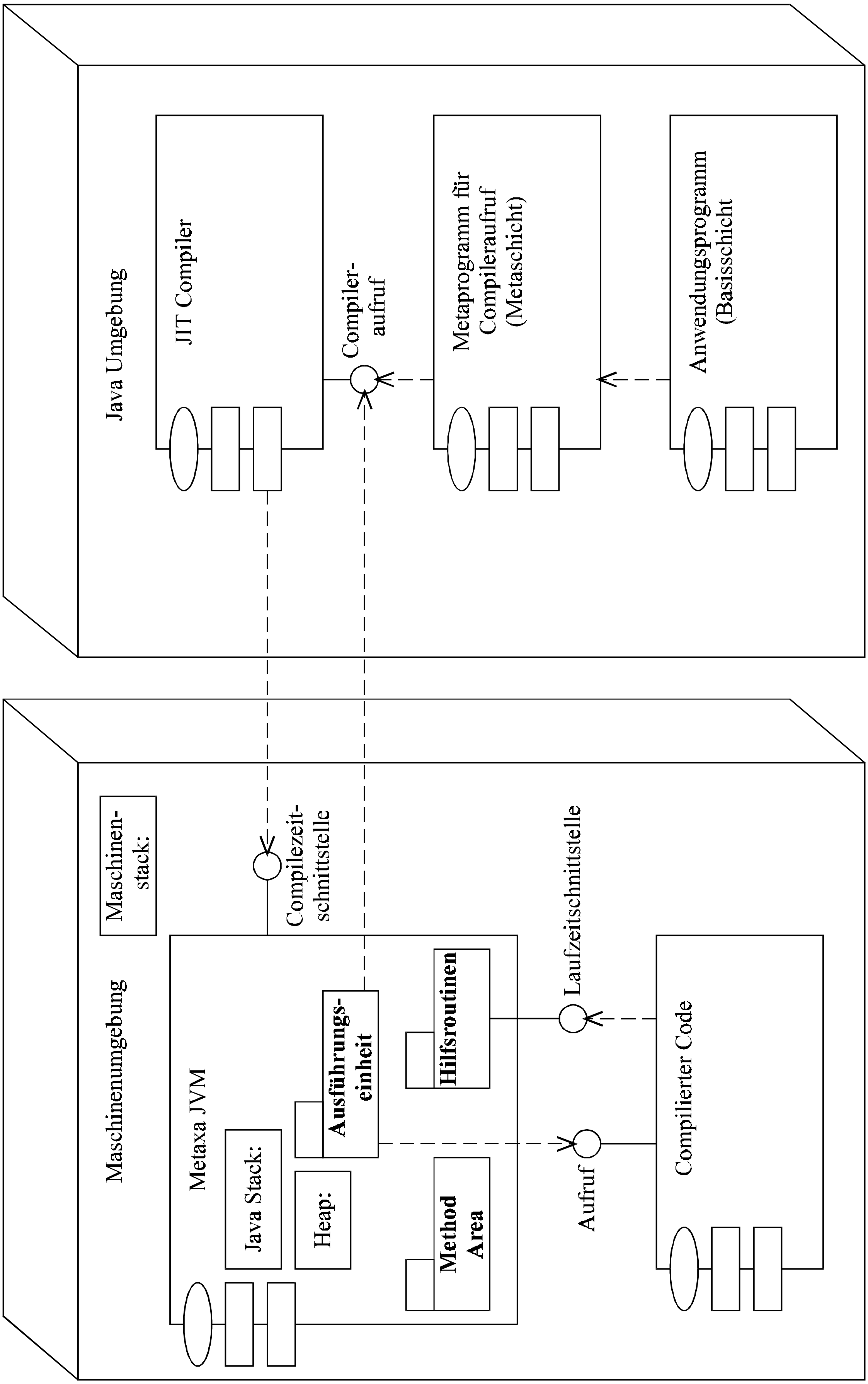

Abbildung 3: Einbettung des Compilers in die Umgebung

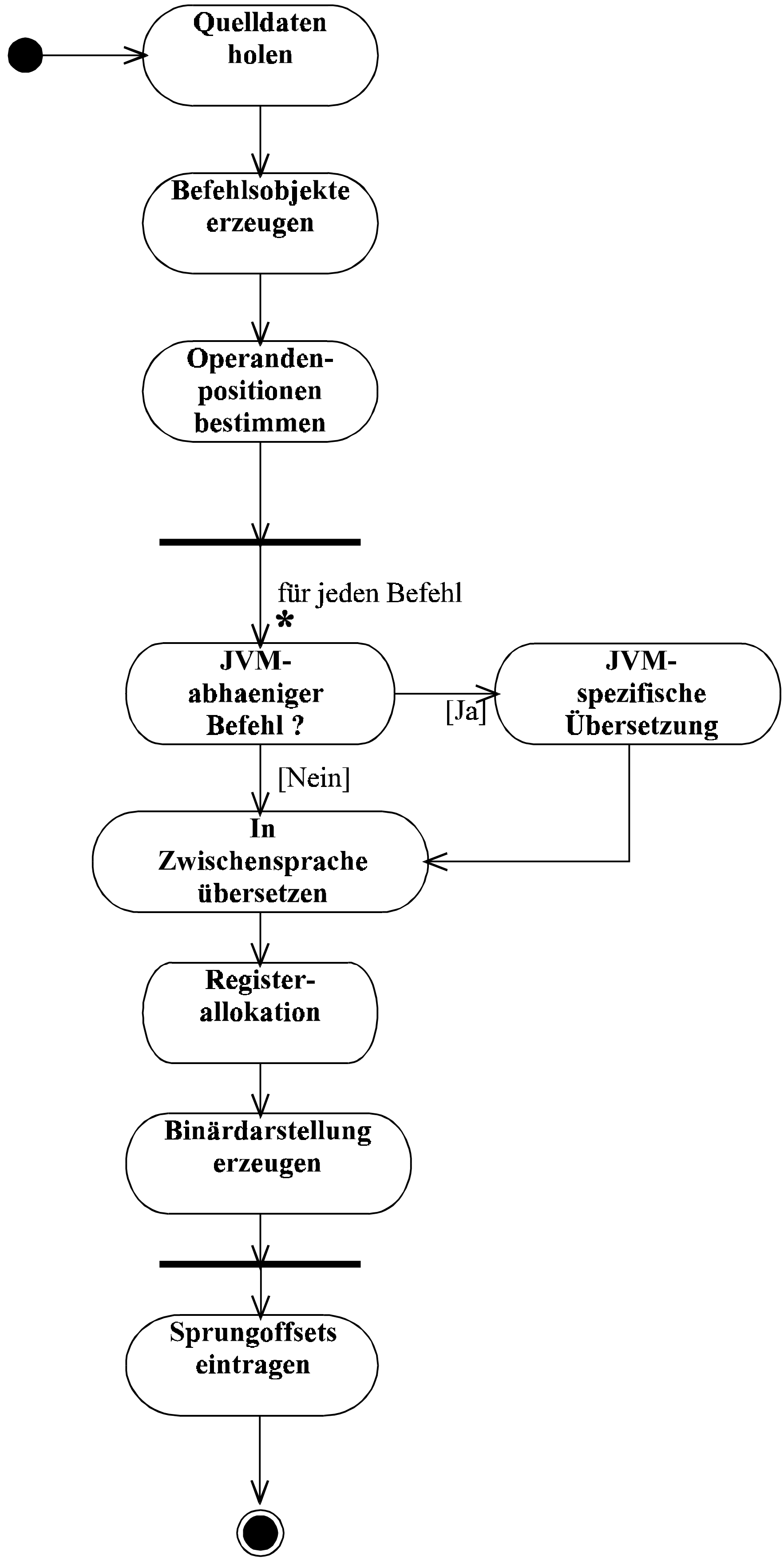

Abbildung 4: Ablauf des Übersetzungsprozesses

6.3 Grobstruktur des Compilers

6.3.1 Komponenten des Compilers

Das Paketdiagramm in Abbildung 5 zeigt die Komponenten des Compilers. Er besteht aus einem Frontend, das von der Zielmaschine weitgehend unabhängig ist (Paket bytecode), und einem Backend, das die maschinenabhängige Codeerzeugung enthält (nativecode, newnative). Außerdem gibt es noch das Paket i386 zur Erzeugung von Binärcode für den Intel 386.

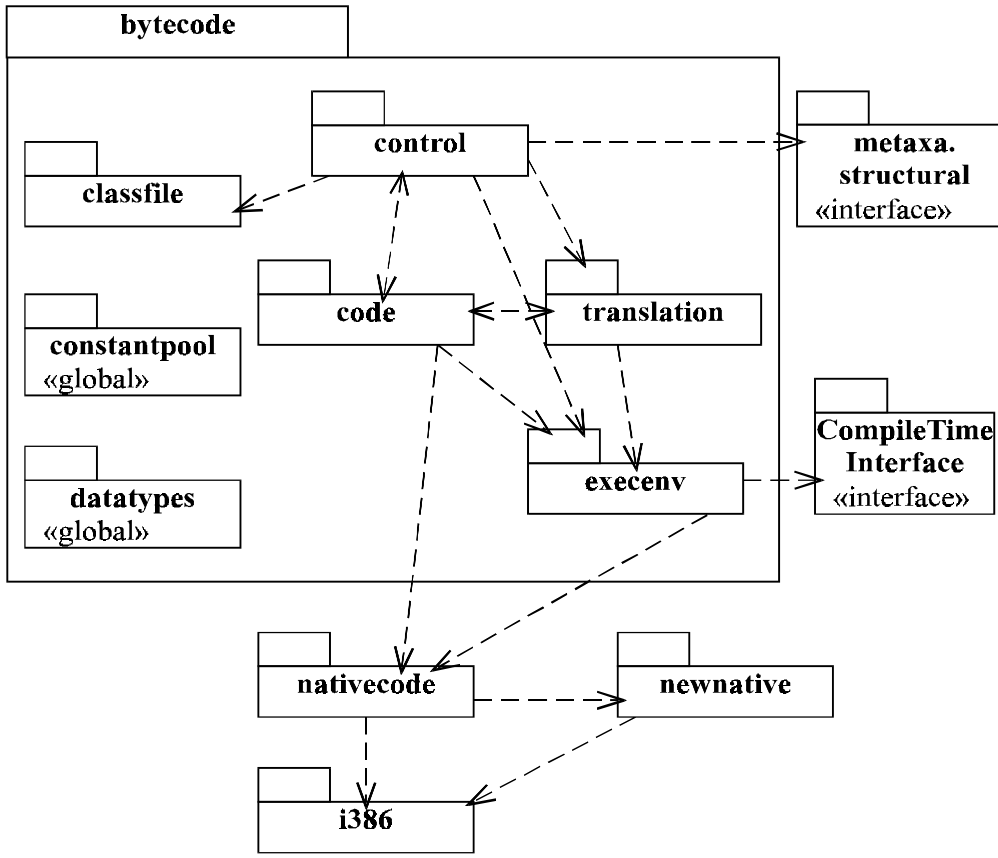

Abbildung 5: Paketdiagramm des Compilers

Die drei letzteren Pakete sind intern relativ einfach aufgebaut, und werden in späteren Kapiteln besprochen. Das erste Paket ist dagegen noch in kleinere Module aufgeteilt.

Dessen zentrales Modul ist dabei das Paket ‘control’, das den gesamten Übersetzungsprozeß steuert. Seine wichtigste Komponente ist eine Klasse die eine einzelne Javamethode darstellt (BCMethodWithCode). Diese Klasse enthält den Code der die Aktivitäten zu Übersetzung dieser Methode kontrolliert.

Das Modul ‘code’ speichert den Bytecode einer Methode in einem speziellen Format. Es enthält auch die Routinen zur Übersetzung eines einzelnen Bytecodebefehls.

Das Modul ‘translation’ arbeitet eng mit dem Modul ‘code’ zusammen. Es enthält spezielle Klassen die verschiedene Operationen auf dem Bytecode ausführen können.

Im Paket ‘execenv’ enthält die Routinen für die Übersetzung von Bytecodebefehlen, die von der jeweiligen JVM abhängig sind. Die Schnittstelle mit all diesen Routinen ist als abstrakte

Basisklasse vorhanden. Für die Implementierung dieser Routinen für eine bestimmte JVM muß man diese Klasse dann ableiten.

Im Paket 'classfile' sind Klassen enthalten, die Klassendateien für die JVM lesen können. Man muß dieses Modul nicht unbedingt verwenden, da der Compiler seine Quelldaten auch aus anderen Quellen bekommen kann.

Das Modul 'constantpool' enthält eine Klasse für den Konstantenpool, und jeweils eine Klasse für jeden möglichen Typ von Konstantenpooleinträgen. Über diese Klassen sind im wesentlichen nur die Daten aus dem Konstantenpool der Klassendatei zugänglich. Die Informationen, die bei der Konstantenpoolauflösung entstehen sind im Augenblick über eine andere Schnittstelle im Modul 'execenv' zu erhalten. Vom Konzept her wäre der Zugriff auf diese Daten über den Konstantenpool auch möglich.

Das Paket 'datatypes' enthält Klassen für die elementaren Datentypen von Java, und für die Signaturen von Feldern und Methoden. Es handelt sich dabei um ein relativ passives Modul.

Über die beiden Schnittstellen 'metaxa.structural' und 'CompileTimeInterface' erhält der Compiler seine Quelldaten von der JVM.

Im Diagramm wurden einige Abhängigkeiten, die zwar im Quellcode vorhanden sind, aber konzeptuell nicht notwendig sind, nicht eingezeichnet.

6.3.2 Ablauf des Übersetzungsprozesses

Die Aktivitäten, die der Compiler zur Übersetzung einer einzelnen Methode ausführen muß, sind in Abbildung 4 ersichtlich. Die Details aus den einzelnen Schritten sind in dem folgenden Kapiteln beschrieben.

Nachdem der Bytecode geladen ist, wird jeder einzelne Bytecodebefehl in ein Objekt übersetzt. Alle Operationen, die im folgenden mit dem Befehl ausgeführt werden müssen, werden von nun an von diesem Objekt übernommen.

Anschließend werden die Positionen der Operanden bestimmt. Diese Schritt wandelt im Prinzip den stackorientierten Bytecode in einen Zweiadreßcode um.

Daraufhin folgt die eigentliche Erzeugung des Zielcodes. Dazu wird der Quellcode vom Anfang bis zum Ende durchlaufen, und jeder Befehl einzeln übersetzt. Handelt es sich dabei um einen Befehl, dessen Übersetzung von der jeweiligen JVM abhängt, wird diese Aufgabe an das Modul 'execenv' delegiert.

Im maschinenabhängigen Teil des Compilers findet nun die Registerallokation statt. Danach kann die Binärdarstellung jedes einzelnen Maschinenbefehls erstellt werden.

Erst nachdem die gesamte Methode übersetzt ist, können die Offsets der Sprungbefehle eingetragen werden, da zuvor die Ziele von Vorwärtssprüngen nicht bekannt sind. Auch andere Konstanten, die zum Zeitpunkt der Codeerzeugung nicht bekannt waren, müssen nun eingetragen werden.

6.4 Darstellung der Quelldaten

Ein JIT-Compiler muß auf die Daten des zu compilierenden Programms zugreifen können. Grob lassen sich diese Daten in zwei Gruppen aufteilen: Die Daten, die die JVM aus der Klassendatei erhalten hat, und die Daten die die JVM zur Laufzeit erzeugt hat (JVM-spezifische Daten).

Beide Arten von Daten kann der Compiler auf unterschiedliche Art und Weise von der JVM erhalten. In diesem Kapitel wird nur auf die Daten der Klassendatei eingegangen.

6.4.1 Aufbau

Bevor der Compiler eine Methode compilieren kann muß er die Quelldaten laden und in eigenen Datenstrukturen speichern.

Dabei hat man die folgenden Komponenten:

- die Methode mit Signatur, Bytecodes und Exceptionhandlern
- den Konstantenpool
- die Klasse zu der die Methode gehört

Zu jeder dieser Komponenten gibt es eine Klasse, die einerseits die jeweiligen Daten speichert, andererseits alle Methoden anbietet, die für die Manipulation der Daten notwendig sind. Letzteres sind im wesentlichen Methoden für die Codeübersetzung.

Abbildung 6 zeigt ein Klassendiagramm dieser Komponenten. Für Methoden gibt es dabei zwei Klassen, eine mit und eine ohne den Bytecode. Letztere wird vorerst nicht für die Übersetzung benötigt. Sie hat nur eine Funktion für das Inlining.

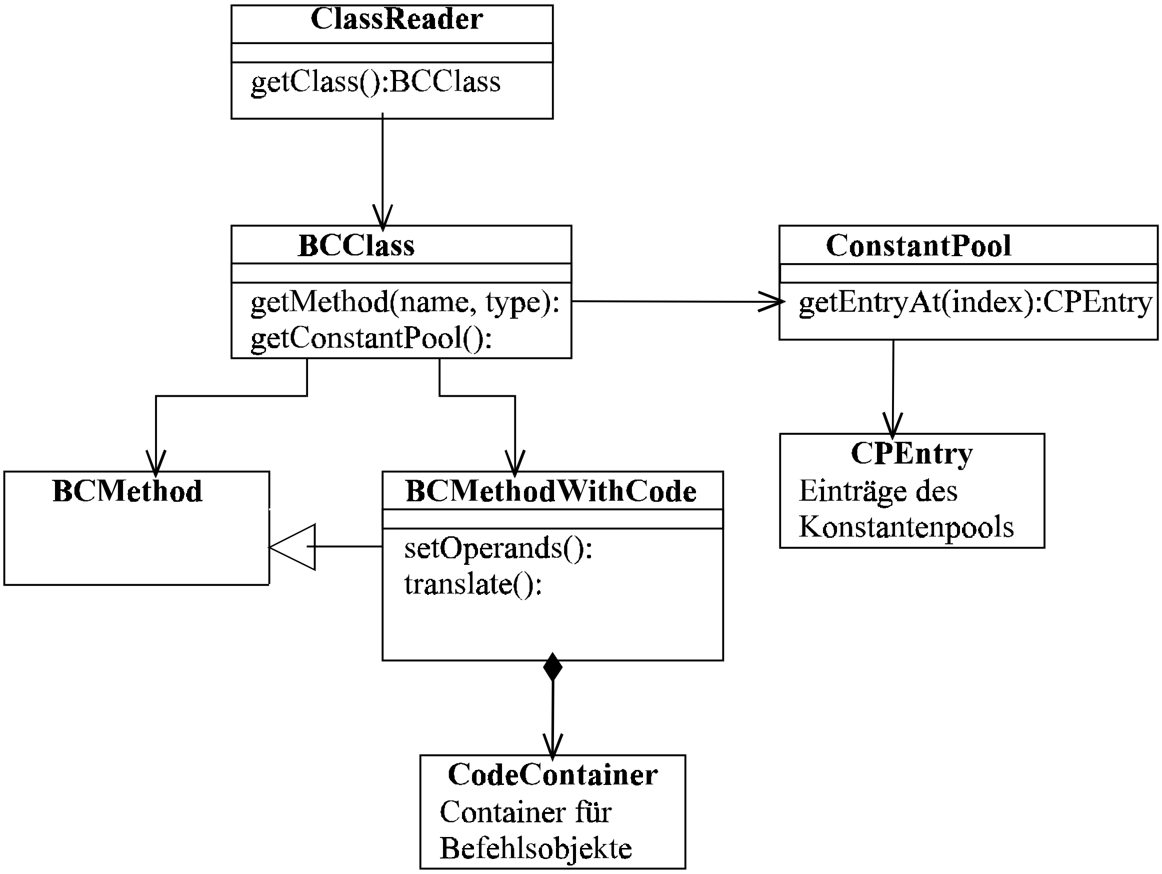

Abbildung 6: Die Klassen zum Speichern von Quellcode

6.4.2 Beschreibung der Klassen

Die Klasse 'BCMethodWithCode' ist dabei am wichtigsten. Ihr Interface enthält die Methoden, die notwendig sind, um Operationen mit dem Bytecode einer Methode auszuführen. Dabei führt diese Klasse die Operationen nicht selbst aus, sondern delegiert sie an andere Klassen. Die Übersetzung einer Methode wird ausschließlich durch das Interface dieser Klasse gesteuert.

Die Klasse BCMethod wird für Methoden verwendet, die von der compilierten Methode aufgerufen werden. Über sie kann der Compiler erfahren, ob es zum Beispiel sinnvoll ist, eine aufgerufene Methode einzubinden (Inlining).

Bei der Klassen 'ConstantPool' handelt es sich um eine Darstellung des Konstantenpools einer einzelnen Klasse. Es ist eine relativ passive Klasse, die nur zum Speichern von Daten verwendet wird. Die einzelnen Einträge des Konstantenpools sind auch durch passive Klassen repräsentiert.

Die Klasse BCClass dient nur dazu, um die Methoden und den Konstantenpool einer Klasse zu erhalten.

Über die Klasse 'ClassReader' kann man die Datenstruktur einer Klasse laden.

6.4.3 Lesen der Daten

Es ist sinnvoll, diese Klassen so zu entwerfen, daß sie aus verschiedenen Quellen initialisiert werden können. So sollte man sowohl das strukturelle Interface von MetaXa, als auch eine einfache Klassendatei als Quelle verwenden können.

Um die Datenstrukturen des Compilers unabhängig von den möglichen Quellen zu halten, wurden die abstrakten Basisklassen ClassSource und MethodSource entworfen. Diese bieten die Methoden, die notwendig sind, um BCMethodWithCode und BCClass zu initialisieren. Für jede mögliche Datenquelle gibt es jeweils eine konkrete Unterklasse. Dieses Vorgehen entspricht dem Entwurfsmuster 'Adapter'.

Für den Konstantenpool wurde eine andere, primitivere Technik für die Initialisierung gewählt, da dies die Struktur der Quelldaten erforderte.

6.5 Die interne Darstellung des Bytecodes

6.5.1 Bytecodebefehle werden durch Objekte dargestellt

Intern werden bei diesem Compiler die Bytecodebefehle durch Objekte dargestellt. Diese Objekte zu erstellen bedeutet zwar zusätzlichen Aufwand, aber dafür lassen sich die folgenden Operationen einfacher implementieren.

Die Klassen für die Befehlsobjekte sind in einer Hierarchie organisiert. Ähnliche Bytecodebefehle wurden dabei jeweils zu einer einzigen Klasse zusammengefaßt. Zum Beispiel werden alle Konvertierungsbefehle von der Klasse BCConversionInstr dargestellt. Auf diese Weise wurden die 201 Bytecodebefehle in ca. 30 Klassen aufgeteilt.

Die Operationen, die bei der Übersetzung mit dem Bytecode ausgeführt werden müssen werden von diesen Befehlsobjekten übernommen. Die Schnittstelle der Basisklasse 'BCInstr' enthält die dazu notwendigen Methoden:

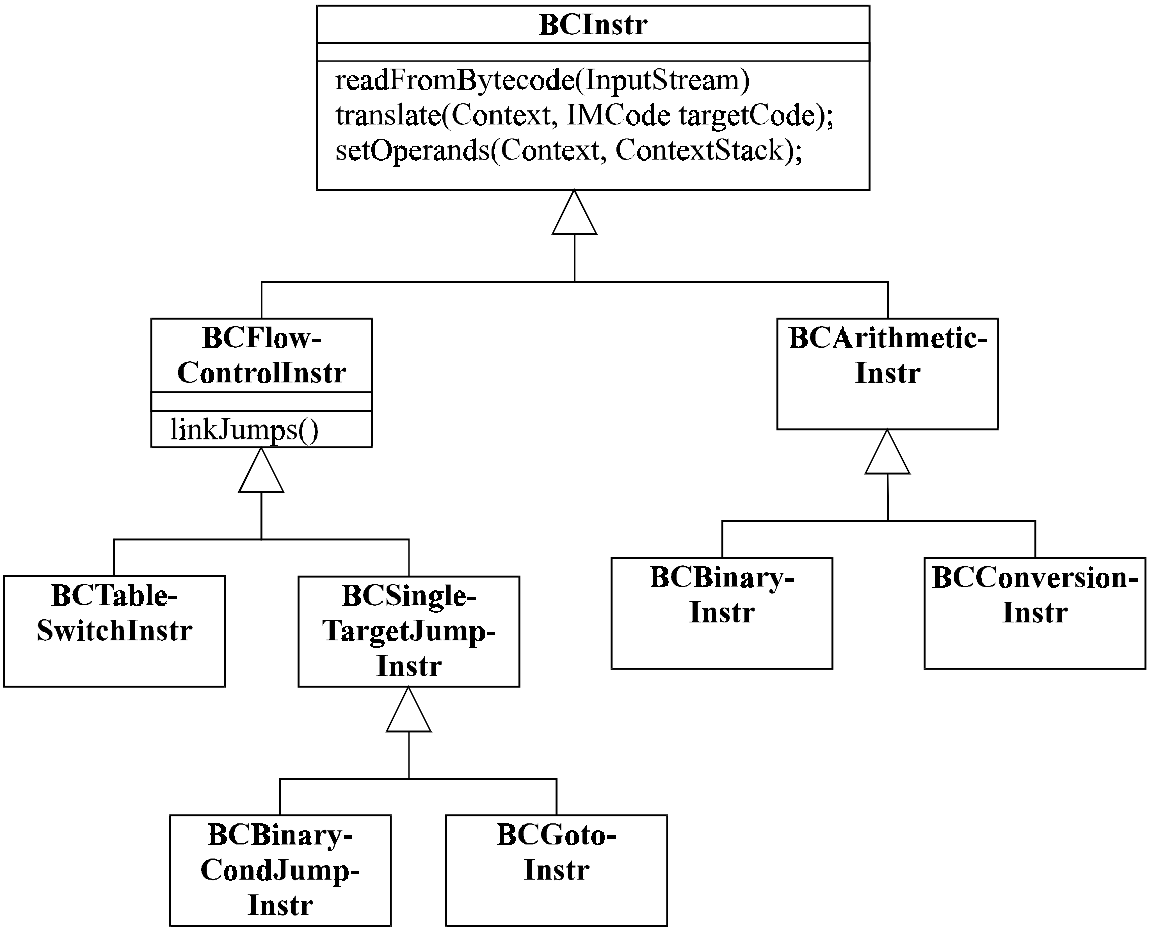

Abbildung 7: Ausschnitt der Klassenhierarchie der Befehlsobjekte

Gespeichert werden die Befehlsobjekte als doppelt verkettete Liste. Da die Befehle meist nur sequentiell durchlaufen werden und das Einfügen von neuen Elementen in die Liste häufig vorkommt, scheint dies die optimale Darstellung zu sein.

Im Gegensatz zum Bytecode kennt jedes initialisierte Befehlsobjekt die Positionen seiner Operanden auf dem Operandenstack. Konzeptuell entspricht diese Darstellung damit einem Zweiadreßbefehlssatz.

Die Befehlsobjekte für Sprungbefehle erfordern etwas mehr Aufwand. Für jeden Sprung wird in die verkettetete Liste an der Stelle des Sprungzieles ein spezielles Objekt ‘JumpTarget’ eingefügt. Das Befehlsobjekte für den Sprungbefehl enthält einen Zeiger auf dieses Objekt. Durch diese Art der Verkettung von Sprüngen lassen sich einige Operationen, wie das Bestimmen der Befehlsoperanden, leichter implementieren.

6.5.2 Erzeugen der Bytecodeobjekte

Im ersten Schritt der Übersetzung wird der Bytecode der Methode in eine doppelt verkettete Liste aus Objekten übersetzt. Jeder Befehl wird dabei durch ein Objekt dargestellt.

Zum Erzeugen der Befehlsobjekte wird das Entwurfsmuster ‘Prototyp’ verwendet. Dazu existiert ein Array, das für jeden Bytecode (Opcode) einen Prototyp für das entsprechende Befehlsobjekt enthält.

Bei der Konvertierung wird dann von jedem Bytecodebefehl der Opcode gelesen und damit über das Array der Prototyp bestimmt. Dieser wird dann geklont. Anschließend kann das geklonte Objekt selbst seine Parameter im Bytecode lesen.

Dieser Prozeß ist relativ langsam, da dabei viele neue Objekte erzeugt werden müssen.

Bevor man die Befehlsobjekte verwenden kann, müssen noch die Sprünge verkettet werden. Dazu muß jedes Befehlsobjekt für Sprünge zuerst die Position des Sprungziels in der verketteten Liste bestimmen, und anschließend ein 'JumpTarget'-Objekt einfügen. Als Hilfsdatenstruktur existiert für diese Operation ein Array, das die Positionen von Bytecodes in der Methode auf Befehlsobjekte abbildet.

6.5.3 Operationen auf dem Bytecode

Alle Operationen, die von Compiler mit einzelnen Bytecodebefehlen ausgeführt werden müssen, werden an die Befehlsobjekte delegiert. Das ist sinnvoll, da die Operationen normalerweise abhängig vom Typ des Befehls sind.

Meist muß eine bestimmte Operation mit allen Befehlsobjekten in einer bestimmten Reihenfolge ausgeführt werden. Zum Übersetzten muß zum Beispiel die Methode translate() aller Befehlsobjekte der Reihe nach vom Anfang bis zum Ende aufgerufen werden. Grundsätzlich ist es aber notwendig, daß die Objekte auch in einer anderen Reihenfolge aufgerufen werden können.

Alle Operationen auf den Bytecodeobjekten sind nach dem gleichen Schema implementiert. Dieses wird durch das Interaktionsdiagramm in Abbildung 8 verdeutlicht.

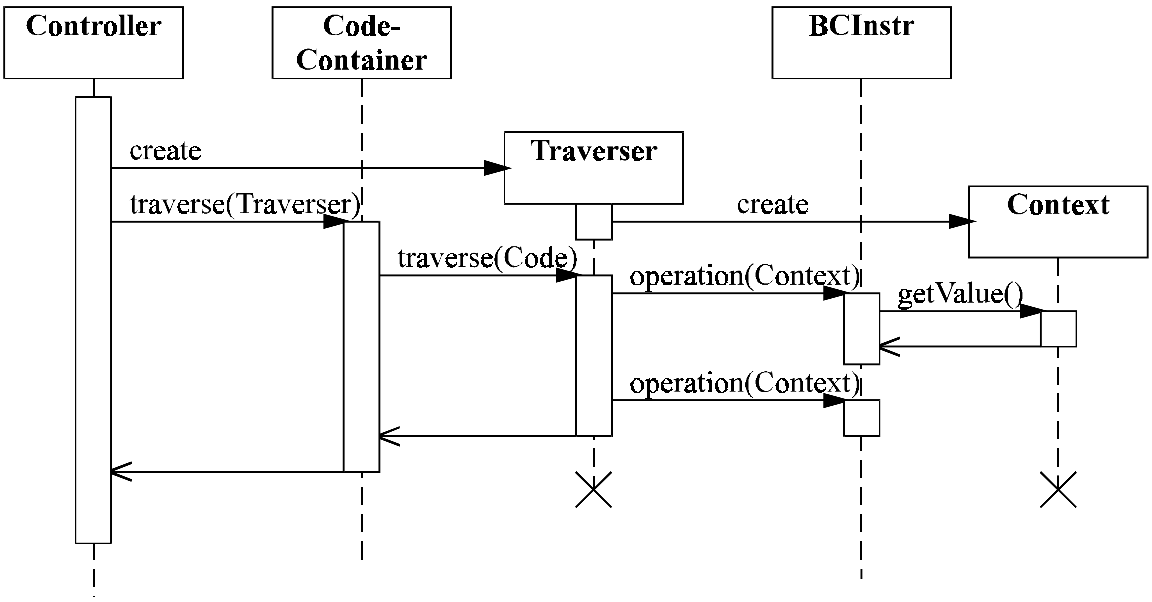

Abbildung 8:Ablauf aller Operationen auf den Befehlsobjekten

Die Klasse des verwendeten Traversers bestimmt dabei die auszuführende Operation. Der CodeContainer speichert die Befehlsobjekte und Exceptionhandler der Klasse. Der Kontext speichert den aktuellen Zustand der Traversierung.

Die Reihenfolge, in der die Befehlsobjekte aufgerufen werden wird ausschließlich durch den Traverser bestimmt. Das ist sinnvoll, da der Traverser auch die auszuführende Operation festlegt, und jede Operation genau eine sinnvolle Traversierungsreihenfolge hat.

Auf diese Art und Weise lassen sich relativ einfach verschiedene Operationen implementieren, wie zum Beispiel:

- Setzen der Operanden
- Verknüpfen der Sprünge
- Erzeugen von nativem Code
- Copypropagation
- Interpretieren (Ausführen der Methode)
- Beseitigen von unnötigen Referencechecks
- Beseitigen toten Codes

Nachteilig bei diesem Entwurf ist, daß jedesmal der gesamte Code durchlaufen wird, auch wenn nur ein Teil der Befehlsobjekte tatsächlich aufgerufen werden muß. Für das Verketten von Sprüngen muß man zum Beispiel nur die Sprungbefehle berücksichtigen. Für derartige Spezialfälle wäre eine eigene optimierte Lösung sinnvoll.

Auch wenn es auf den ersten Blick so aussieht, verwendet dieser Entwurf nicht das Besucher-Entwurfsmuster. Denn für jede neue Operation müssen hier im allgemeinen die Befehlsobjekte erweitert werden, was beim Entwurfsmuster 'Besucher' nicht der Fall ist. Es wäre grundsätzlich möglich, hier das Besuchermuster zu verwenden. Dadurch ließe sich die Implementierung von neuen Operationen deutlich vereinfachen. Die Folge wäre allerdings ein deutlich langsameres Programm. Aus diesem Grund wurde dieser einfachere Ansatz gewählt.

6.6 Schritte der Übersetzung

Eine Übersicht über die einzelnen Schritte der Übersetzung findet man in Kapitel 6.3. Hier soll nur auf die Details eingegangen werden.

Der erste Schritt ist eigentlich die Erzeugung der Bytecodeobjekte. Dieser wurde schon in Kapitel 6.4 beschrieben.

6.6.1 Bestimmen der Operandenpositionen

Die JVM ist stackorientiert. Das bedeutet, daß die meisten Bytecodebefehle ihre Argumente vom Operandenstack holen, und ihre Ergebnisse auf den Operandenstack zurückschreiben. Das erlaubt es, einen Interpreter besonders einfach und effizient zu schreiben.

Für die Übersetzung des Bytecode in nativen Code ist es aber besser, wenn man die absoluten Positionen der Operanden eines jeden Befehls kennt. Dadurch kann der erzeugte Code etwas effizienter werden, da man den Stackzeiger nicht verwalten muß.

Auch wird die Implementierung der Registerallokation und andere Aufgaben dadurch deutlich vereinfacht.

Die Bestimmung der Position des Stackzeigers für einen Befehl einer Methode ist nicht ganz einfach, da sie im Prinzip von allen Befehlen der Methode abhängt, die zuvor ausgeführt wurden.

Um nun die Stackpositionen der Operanden zu bestimmen, durchläuft man den gesamten Code der Methode in derselben Reihenfolge, wie der auch bei einer möglichen Ausführung durchlaufen wird. Dabei wird für jeden Befehl die Position der Operanden gespeichert, und der neue Wert des Stackzeigers berechnet.

Im Prinzip wird in diesem Schritt der stackorientierte Bytecode in eine Zweiadreßcode übersetzt.

Im selben Schritt werden bei diesem Compiler auch noch die Indizes auf die Einträge des Konstantenpools durch Referenzen auf diese Einträge ersetzt.

Die Implementierung folgt dem im Kapitel 6.5.3 beschriebenen Schema.

6.6.2 Übersetzen der Befehle

Die beim eigentlichen Übersetzen notwendigen Aufgaben werden eigentlich zum Großteil im maschinenabhängigen Teil des Compilers ausgeführt. Die Befehlsobjekte rufen nur die entsprechenden Methoden dessen Interfaces auf.

Befehle, deren Übersetzung von der verwendeten virtuellen Maschine abhängen werden nicht direkt übersetzt. Deren Übersetzung wird an eine andere Klasse delegiert (Kapitel 6.7). Auf diese Weise sind die Befehlsobjekte unabhängig von der jeweiligen JVM. Das betrifft im wesentlichen Zugriffe auf Felder und Methodenaufrufe, sowie Befehle, die Hilfsfunktionen der JVM benötigen.

6.7 Die JVM-abhängige Übersetzung

6.7.1 Die Schnittstelle

Manche Bytecodebefehle können nur übersetzt werden, wenn man einige Eigenschaften der JVM, unter der der Code ausgeführt werden soll, kennt. Das betrifft die folgenden Befehle:

- Zugriff auf Felder und Arrays (getfield ...)
- Aufruf von Methoden (invokevirtual ...)
- alle Befehle, die Hilfsroutinen der JVM aufrufen (instanceof ...)
- Befehle die Ausnahmen auslösen (athrow)
- Tests, ob eine Ausnahme ausgelöst werden muß (checkcast, Test ob Referenz Null ist)

Außerdem ist die Behandlung der folgenden Punkte von der JVM abhängig:

- die Aufrufkonvention der compilierten Methode
- das Stacklayout der compilierten Methode
- die Aufrufkonvention von Methoden und Hilfsroutinen
- die Art der Ausnahmebehandlung (das, was passiert, wenn eine Ausnahme aufgetreten ist)

All diese Aufgaben werden an die Klasse ExecEnvironment delegiert. Dabei handelt es sich um eine abstrakte Basisklasse. Für jede JVM muß eine konkrete Unterklasse davon gebildet werden, deren Methoden die Codeerzeugung übernehmen.

Abbildung 9 zeigt eine repräsentative Teilmenge der Methoden dieser Klasse.

ExecEnvironment
getInitialStackFrame(numLocalVariables,...):StackFrame addProlog(targetCode:IMCode, stackFrame,...) addEpilog callVirtualMethod(..., methodRefCPEntry, objectRefSlot, ...); getField(targetCode, stackFrame, fieldRefCPEntry, datatype, objRefSlot, returnValueSlot); newObject throwException checkIntNotZero checkReference instanceOf longDiv

Abbildung 9: Einige Methoden von ExecEnvironment

Alle Methoden bis auf getInitalStackFrame() dienen dabei ausschließlich der Codeerzeugung.

Alle Codeerzeugungsroutinen erhalten als Parameter das Objekt für den Zielcode, und den aktuellen Stackframe. Den Stackframe benötigen sie, da mache Routinen die Position der temporären und reservierten Variablen kennen müssen.

6.7.2 Das Stacklayout

Jede compilierte Methode benötigt eine Laufzeitumgebung (Stackframe). Das ist der Speicherbereich, in dem sie ihre lokalen Variablen und die Stackoperanden, aber auch andere Daten wie die Rückkehradresse der Methode speichert. In dieser Implementierung wird der Stackframe als zusammenhängendes Array von 32-Bit Worten (Zellen, Slots) betrachtet. Jede Zelle wird dabei durch einem von Null anfangenden Index angesprochen. Daten mit 64 Bit werden in zusammenhängenden Zellen gespeichert. Diese Form der Stackframes ist den Stackframes der JVM sehr ähnlich.

Das Stacklayout beschreibt, wie die Daten im Stackframe gespeichert werden. Zum Beispiel beschreibt es, wo die lokalen Variablen liegen. Es ist sinnvoll, das Stacklayout nicht im Compiler fest zu verdrahten, da es von dem JVM abhängig sein kann.

Wenn man zum Beispiel die Stackframes der compilierten Methoden nicht auf dem Maschinenstack sondern auf dem Javastack allokiert, ändert sich mit Sicherheit auch das Stacklayout.

Das Stacklayout wird durch die abstrakte Klasse Stackframe im Paket bytecode.execenv dargestellt. Die Unterklassen dieser Klasse realisieren jeweils ein bestimmtes Stacklayout.

Jede konkrete Unterklasse von StackFrame beschreibt jeweils die Anzahl und Lage der folgenden Variablentypen für eine zu compilierende Methode:

- lokale Variablen (JVM)
- Stackoperanden (JVM)
- reservierte Variablen (für den Compiler)
- temporäre Variablen (für den Compiler)

In reservierten Variablen werden dabei Werte gespeichert, die während der gesamtem Methode gültig sein müssen. Die temporären Variablen sind Hilfsvariablen, die während der Codeerzeugung für einen einzelnen Befehl gültig sein können.

Da die zu verwendende Unterklasse von StackFrame von der Klasse ExecEnvironment abhängt, werden die Objekte des Typs StackFrame von der Fabrikmethode [Gamma u.a. 95] ExecEnvironment.getInitalStackFrame() erzeugt.

In der aktuellen Version enthält jeder Stackframe die vier Variablentypen in der folgenden Reihenfolge:

- lokale Variablen
- reservierte Variablen
- temporäre Variablen
- Stackoperanden

Dabei gibt es zwei reservierte Variablen, eine enthält den Zeiger auf das Java Native Interface (JNI), die andere speichert das Kontrollwort des Fließkommaprozessors. Der Zeiger auf das JNI wird von der compilierten Methode nur für den Aufruf von Hilfsfunktionen und für die Ausnahmebehandlung gebraucht.

6.7.3 Das Laufzeitschnittstelle zu MetaXa

Über die Laufzeitschnittstelle kann der compilierte Code mit der JVM kommunizieren. Die Schnittstelle ist spezifisch für die JVM von MetaXa. Daher ist der gesamte Zugriff auf diese Schnittstelle im der Klasse execenv.ExecEnvironmentMeta gekapselt. Der Rest des Compilers ist davon nicht abhängig.

Die Laufzeitschnittstelle umfaßt Routinen für die folgenden Aufgaben:

- Aufruf von (interpretierten) Methoden
- Allokieren neuer Objekte und Arrays
- Werfen von Ausnahmen
- Finden eines Exceptionhandlers für eine geworfene Methode
- Routinen, die für die Ausführung der folgenden Bytecodes benötigt werden:
    - instanceof
    - checkcast
    - monitorenter, monitorexit
    - ldiv, lrem, lcmp, lshr, lshl, lushr

Ein Beispiel für eine Hilfsroutine ist das Allokieren des Speichers für Objekte:

```
int allocObjectMemory(JNIEnv *env, int classID);
```

Der erste Parameter dient der JVM, um den aktuellen Thread zu identifizieren. Er wird dem compilierten Code von der JVM übergeben, und der verwendet ihn nur als Parameter für die Hilfsroutinen und für die Ausnahmebehandlung. Die meisten Hilfsroutinen benötigen diesen Parameter.

Der zweite Parameter identifiziert die Klasse, für deren Instanz die Routine Speicher belegen soll. Der Rückgabewert ist eine Referenz auf ein noch nicht initialisiertes Objekt.

Die anderen Hilfsroutinen haben einen ähnlichen Aufbau.

6.7.4 Die Compilezeit-Schnittstelle

Die Compilezeit-Schnittstelle dient dem Compiler dazu, statische Daten der JVM zu erhalten. Diese Daten werden meist als Konstanten in den Code eingefügt.

Die statischen Daten, die der Compiler über die Compilezeit-Schnittstelle bekommt, lassen sich aufteilen in die Konstanten der JVM und in die Werte, die von Konstantenpooleinträgen abhängen.

Die Konstanten der JVM sind meist die Offsets von Feldern in den internen Strukturen der JVM. Um die Länge eines Arrays zu erhalten, muß der Compiler zum Beispiel das Offset des Längenfeldes in der Arraystruktur wissen.

Interessanter ist der zweite Typ von statischen Daten. Dazu zählen die folgenden Werte:

- die ID von Methoden (siehe letztes Kapitel)
- der Index einer Methode in einer virtuellen Methodentabelle
- die ID von Klassen
- der Offset der Felder von Javaobjekten
- der Index von Methoden in der virtuellen Methodentabelle
- die Adresse von statischen Feldern
- die Adresse von Stringobjekten des Konstantenpools

Die Werte hängen von den Konstantenpooleinträgen ab. Daher muß der Compiler auch die Daten dieser Einträge den Methoden der Compilezeit-Schnittstelle übergeben. Um zum Beispiel die ID einer Methode zu erhalten, muß er den Namen, die Signatur und Klasse der Methode übergeben. Alternativ dazu könnte man die Werte auch über den Index des Konstantenpooleintrags identifizieren.

All die obigen Werte können erst durch die Konstantenpoolauflösung ermittelt werden. Daher muß sie bei dieser Schnittstelle zur Compilezeit stattfinden. Nach dem JVM Reference Manual ist das erlaubt [Lindholm u.a. 96, Kapitel 2.16.3, Linking].

6.7.5 Der Aufruf von Methoden

In einer JVM gibt es vier verschiedene Arten von Methodenaufrufen:

- virtuelle Methodenaufrufe
- nichtvirtuelle Methodenaufrufe
- statische Methodenaufrufe

- Aufrufen über Interfaces

Man muß unterscheiden, ob bei einem Methodenaufruf die Kontrolle an die JVM übergeben werden soll, oder ob eine bereits compilierte Methode direkt aufgerufen werden soll.

Im ersten Fall wird zuerst eine Hilfsroutine der JVM aufgerufen, die den eigentlichen Methodenaufruf ausführt. Es ist für den compilierten Code transparent, wie die JVM diesen Methodenaufruf durchführt. Sie kann die Methode interpretieren, erst compilieren und dann ausführen, oder compilierten Code direkt ausführen. In jedem Fall muß der compilierte Code der Hilfsroutine mitteilen, welche Methode ausgeführt werden soll. Daher hat diese Routine folgende Signatur (C-Syntax):

```
int invokeMethod(JNIEnv *env, int methodID, <Parameter der Javamethode>);
```

Der Parameter ‘methodID’ identifiziert dabei die auszuführende Methode. Der Rückgabewert dieser Routine ist der Rückgabewert der aufgerufenen Javamethode. Für die verschiedenen Arten von Methodenaufrufen und für verschiedene Rückgabewert gibt es jeweils eine eigene Hilfsroutine.

Der Aufruf von compilierten Methoden über diese Hilfsroutine ist ineffizient. Ein direkter Aufruf ist hier besser. Dabei ist allerdings zu beachten, daß zur Compilezeit die aufgerufenen Methode noch nicht compiliert sein müssen. Man muß daher den Methodenaufruf zuerst als Aufruf an die Hilfsroutine compilieren, und kann ihn erst später durch einen direkten Aufruf von compilierten Code ersetzen.

Damit dieses Ersetzen möglichst einfach ist, wurde in dieser Implementierung die Aufrufkonvention der compilierten Methoden identisch mit der Aufrufkonvention der obigen Hilfsroutine gewählt. Auch compilierte Methoden bekommen einen Parameter ‘methodID’ übergeben, obwohl sie diesen überhaupt nicht verwenden.

Bei Aufrufen an virtuelle Methoden und Methoden von Interfaces muß die Adresse der aufzurufende Methode zur Laufzeit ermittelt werden, da sie vom Typ des Empfängers abhängt. Bei virtuellem Methoden erhält man die Adresse über eine virtuelle Methodentabelle. Der Compiler muß daher entsprechenden Code erzeugen. Auch Aufrufe von noch nicht compilierten Methoden können über eine virtuelle Methodentabelle erfolgen, wenn man sie mit Zeigern auf die Hilfsroutine zum Methodenaufruf initialisiert. Der direkte Aufruf von Interfacemethoden ist in diesem Compiler noch nicht implementiert.

Bei statischen und nichtvirtuellen Aufrufen ist die Zielmethodenadresse als Konstante in den compilierten Code eingebaut.

Die Aufrufkonvention für compilierten Code betrifft nur den Prolog und den Epilog der Methode. Da beide in den Unterklassen von ExecEnvironment erzeugt werden (Methoden addProlog() und addEpilog()), lassen sie sich relativ einfach ändern.

6.7.6 Die korrekte Initialisierung von Klassen

Nach dem JVM Reference Manual muß die Initialisierung einer Klasse bei ihrer ersten aktiven Verwendung stattfinden.

Für den compilierte Code bedeutet das, daß er bei machen Ereignissen eine Initialisierung auslösen muß. Für Methodenaufrufe und Objektallokationen ist das in der aktuellen Implementierung kein Problem des Compilers, da diese Aktionen von der JVM ausgeführt werden. Einzig die statischen Feldzugriffe finden im compilierten Code statt, und können eine Initialisierung auslösen.

Es scheint sinnvoll, hier die Lösung von Sun (Kapitel 5.2.4) zu übernehmen. Man fügt vor dem Feldzugriff einen Aufruf einer Hilfsroutine ein, die die Initialisierung auslöst. Nach der Initialisierung wird der Aufruf entfernt (z.B. durch NOPs ersetzt). Das ist aber noch nicht implementiert.

6.7.7 Nachträgliche Modifikation des compilierten Codes

Für manche Zwecke ist es sinnvoll, den compilierten Code nach dem Installieren noch zu verändern.

Das trifft für die Entfernung der Hilfsroutine aus dem letzten Kapitel zu. Wichtig ist es aber insbesondere, um Methodenaufrufe umzulenken. Nach dem Compilieren einer Methode sollte man zum Beispiel alle Aufrufe der Methode über die JVM durch direkte Aufrufe des compilierten Codes ersetzen. An diesem Beispiel sollte eine mögliche Implementierung des selbstmodifizierenden Codes diskutiert werden. Sie ist im Compiler noch nicht realisiert.

Zwei Fragen sind dabei zu klären: Bei welchen Ereignis soll der Code modifiziert werden, und welche Stelle muß wie ersetzt werden.

Die erste Frage ist ein Problem der JVM. Nur diese kann die jeweiligen Ereignisse überwachen. Das Umlenken eines Methodenaufrufs kann zum Beispiel dann passieren, wenn genau dieser Aufruf ausgeführt wird, oder nachdem die aufgerufene Methode compiliert wurde. Für die Implementierung in Java kann man dafür Listener verwenden.

Die zweite Frage ist ein Problem des Compilers. Dieser kennt die Position der Methodenadresse im Code. Außerdem weiß er, ob die Adresse als relative oder absolute Adresse eingetragen werden muß. Diese Informationen kann er in einer Klasse kapseln. Objekte dieser Klasse können dann die Modifikation des Codes durchführen.

Zusammen mit dem compilierten Code muß der Compiler nun auch diese Objekte installieren. Die JVM erstellt dann entsprechende Listener, so daß die Methodenaufrufe des compilierten Codes umgelenkt werden können, wenn die aufgerufenen Methoden compiliert sind.

6.7.8 Ausnahmen

Zwei Aspekte sind bei der Ausnahmebehandlung zu beachten: Wie werden Ausnahmen ausgelöst, und wie werden sie behandelt. Der erste Aspekt betrifft die Übersetzung aller Bytecodes, die Ausnahmen auslösen können. Der zweite Aspekt betrifft zusätzlich noch die Übersetzung der Methodenaufrufe und natürlich die Exceptionhandler der Methode.

Zwei Alternativen zur Ausnahmebehandlung wurden schon in Kapitel 5.2.7 besprochen. Hier wird nur die Kapselung der Ausnahmebehandlungsroutinen und die konkrete Implementierung für die MetaXa JVM behandelt.

6.7.8.1 Kapselung der Codeerzeugung für Ausnahmen

Es gibt viele Bytecodebefehle die eine Ausnahme auslösen können, wenn nicht erlaubte Operanden verwendet werden. Zum Beispiel löst ein Feldzugriff mit ‘getfield’ eine NullPointerException, wenn die verwendete Referenz Null ist. Der Compiler muß für solche Bytecodebefehle zuerst Code erzeugen, der die Operanden überprüft, und kann erst dann die eigentliche Operation ausführen. Diese beiden Schritte werden in ExecEnvironment von zwei getrennten Methoden übernommen. Zum Beispiel bei 'getfield' wird zuerst mit der Methode ExecEnvironment.checkReference() der Code für den Test auf Nullreferenzen erzeugt, und dann mit der Methode ExecEnvironment.getField() der Code für den Feldzugriff. Diese Aufteilung hat den Vorteil, daß auch ein optimierender Compiler, der unnötige Tests entfernt, diese Schnittstelle verwenden kann.

6.7.8.2 Die konkrete Implementierung

Die konkrete Implementierung der Ausnahmebehandlung orientiert sich bei diesem Compiler an der Methode, die in Cacao verwendet wurde [Grafl 97], (Kapitel 5.2.7). Sie ist noch nicht vollständig fertiggestellt.

Wenn eine Ausnahme auftritt, so wird bei MetaXa ein Zeiger auf das geworfene Exception-Objekt in einer speziellen Struktur (JNIEnv*) gesetzt. Pro Thread gibt es genau eine solche Struktur. Das Setzen dieser Variable erfolgt dabei immer durch die Hilfsroutinen der JVM.

Nach dem Aufruf einer Methode muß der compilierte Code nun immer überprüfen, ob diese Variable gesetzt ist. Ist sie es, so hat die aufgerufene Methode eine Ausnahme ausgelöst, und er compilierte Code muß diese Ausnahme behandeln.

Wenn festgestellt wurde, daß eine Ausnahme aufgetreten ist, so muß getestet werden, ob die Ausnahme in der Methode einen Handler hat. Wenn das so ist, so muß der Handler aufgerufen werden, ansonsten wird die Methode sofort beendet.

Für den Test, ob für einen Handler eine geworfene Ausnahme existiert gibt es in der MetaXa JVM eine Hilfsroutine. Diese bekommt den aktuellen Befehlszeiger und eine ID für die Methode, und liefert entweder die Adresse der Behandlungsroutine, oder Null, wenn keine existiert. Die Implementierung der letzten beiden Schritte ergibt sich damit von selbst.

Der Vorteil dieser Implementierung ist, daß sie relativ gut portierbar ist, da die JVM keine Stackbereinigung des Maschinenstacks durchführen muß. Allerdings ist sie weniger effizient. Zum Beispiel ist es mit ihr nicht möglich, Ausnahmen durch Hardwaresignale auslösen zu lassen.

6.8 Der maschinenabhängige Teil

Der maschinenabhängige Teil des Compilers ist vollkommen unabhängig vom maschinenunabhängigen Teil. Er besteht aus den Paketen 'nativecode' und 'i386'. Von der Struktur her ist dieser Teil eher funktional als objektorientiert realisiert. Abbildung 10 zeigt ein Klassendiagramm mit den wichtigsten Klassen.

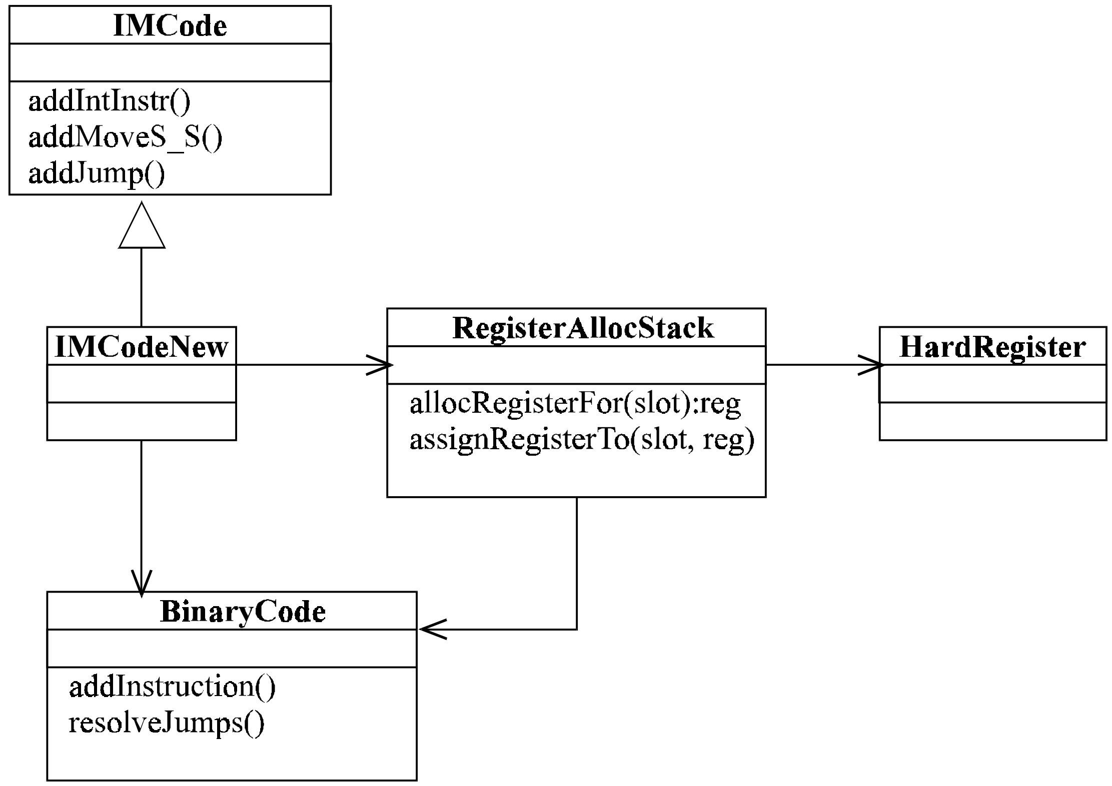

Abbildung 10: Die wichtigsten Klassen des maschinenabhängigen Teils

6.8.1 Die Schnittstelle

Von besonderer Bedeutung ist die Schnittstelle der Klasse IMCode. Sie enthält eine große Anzahl von Methoden, die alle eine ähnliche Semantik haben: Sie erstellen den Maschinencode einer bestimmten Operation. All diese Befehle haben den Präfix 'add', da der neue Code zum bereits erstellen Maschinencode hinzugefügt wird.

Die folgende Klassendeklaration zeigt einen repräsentative Untermenge der 'Codeerzeugungsmethoden' von IMCode.

```
class IMCode {
  // Kopieren zwischen Slots
void addMoveIntS_S(int dstSlot, int srcSlot);
  // Laden einer Hauptspeicherzelle in den Slot
void addMoveIntS_MB(int dstSlot, int baseSlot, int offset);
  // arithmetische Operation (operator == ADD, SUB ...)
void addInstrIntS_S(int operator, int dstSlot, int srcSlot);
  // Konvertieren von Datentypen
  // (dstType/srcType == INT, LONG ...)
void addInstrCnvS(int slot, int dstType, int srcType);
  // Einfügen eines Sprungs
void addJump(UnresolvedJump jumpObject);
  // Einfügen eines Sprungziels
void addJumpTarget(UnresolvedJump jumpObject);
  // Einfügen eines bedingten Sprungs
  // (condition == EQUAL, LESS, ...)
void addCondJumpIntS_S(int condition, int slot1, int slot2,
                       UnresolvedJump jumpObject);
}
```

Konzeptuell entspricht diese Schnittstelle einer eigenen Zwischensprache bei der Übersetzung. Der Name der Codeerzeugungsmethode entspricht dem Befehlsnamen, die Parameter der Methode den Operanden des Befehls.

Die Aspekte, die man bei der Wahl des Befehlssatzes einer solchen Sprache beachten sollte, sind in [Davidson u.a. 84b] beschrieben. Insbesondere sollte man die Zwischensprache an den Befehle der möglichen Zielmaschinen anpassen. Denn nur so ist es möglich, ohne maschinenabhängige Optimierungen einigermaßen effizienten Code zu erzeugen. Das bedeutet, daß die Befehle der Zwischensprache eher mächtiger sein sollten wie die Zielsprache.

Die hier gewählte Sprache ist unabhängig von der Zielmaschine. Prinzipiell ist es also möglich, den Compiler zu portieren, indem man nur dieses Backend neu implementiert. Allerdings ist die Zwischensprache aufgrund der obenstehenden Überlegungen am Befehlssatz des Intel 386 orientiert.

Die Operanden der Befehle sind die Indizes der Zellen (Slots) des aktuellen Stackframes. Die Indizes beginnen mit 0. Die Gesamtzahl der Zellen wird dem Objekt der Klasse IMCode bei der Konstruktion mitgegeben.

Für die Erzeugung des Maschinencodes wird zuerst ein Objekt des Typs IMCode erstellt. Dieses erzeugt die Datenstrukturen in denen der compilierte Code gespeichert wird. Anschließend werden die ‘Codeerzeugungsmethoden’ aufgerufen, wodurch der compilierte Code erstellt wird. Wenn das für alle Befehle der Methode getan ist, kann man die Sprungoffsets eintragen lassen, und sich den compilierte Code als Bytearray geben lassen.

6.8.2 Die Registerallokation und der virtueller Stack

Grundsätzlich werden bei diesem Compiler alle von einer Methode benötigten Variablen auf den Maschinenstack allokiert. Die Position einer Variable auf dem Stack wird dabei durch ihre Nummer bestimmt. Nur während der Ausführung eines elementaren Blocks werden einige dieser Variablen in Register ausgelagert. Dieses Vorgehen ist sinnvoll, da der Intel 386 über zuwenige Register für eine statische Registerzuordnung verfügt.

6.8.2.1 Der virtuelle Stack

In dieser Implementierung wird bei der Registerallokation und für die Bestimmung der optimalen Typen ein virtueller Stack verwendet.

Der implementierte virtuelle Stack folgt im wesentlichen der Beschreibung aus Kapitel 5.2.6.1. Er besteht aus einem Array aus Referenzen das dieselbe Größe hat wie der aktuelle Stackframe. Jedes Feld des Arrays beschreibt, wo der Wert der entsprechenden Zelle gespeichert ist. Entweder ist der Wert im Hauptspeicher, oder in einem Register, oder es handelt sich um eine Konstante. Im ersten Fall ist das entsprechende Feld Null, ansonsten zeigt es auf ein Objekt, das die ID des Registers bzw. den Wert der Konstante speichert (Klasse HardRegister).

Die Schnittstelle des virtuellen Stacks erlaubt es zum Beispiel, für eine Zelle ein Register zu allokieren, oder einer Zelle ein Register zuzuweisen bzw. den Wert der Zelle in den Speicher zurückschreiben. Die folgende Klassendeklaration zeigt einige Methoden des virtuellen Stacks (der hier RegisterAllocStack heißt).

```
class RegisterAllocStack {
  // Slot in Register auslagern, Returnwert ist die ID des
  // Registers
int allocRegisterFor(int slot);
  // Register allokieren, das den Wert des Slots enthält, das
  // aber nicht dem Slot zugeordnet ist
int allocFreeRegisterFor(int slot);
   // Slot in Speicher zurückschreiben
void freeSlot(int slot);
   // einem Slot ein Register zuweisen (Der Slot übernimmt den
   // Wert des Registers)
void assignRegisterTo(int slot, int register);
   // liefert einen Operanden für den Zugriff auf einen Slot
   // (es kann sich dabei um einen Register-, Speicher- oder
   // um einen konstanten Operanden handeln)
InstrOperand slotOpConst(int slot, InstrOperand op);
}
```

Die folgende Methode verwendet diesen Stack. Sie erzeugt Code, um die Werte in den Slots srcSlot1 und srcSlot2 zu addieren, und das Ergebnis nach Slot dstSlot zu schreiben.

```
void makeCodeIntAdd(int dstSlot, int srcSlot1, int srcSlot2) {
  // ersten Operanden in freies Register kopieren
int srcReg1 = virtualStack.allocFreeRegisterFor(srcSlot1);
  // Code erzeugen
targetCode.addInstr(IntInstr.ADD, i1.register(srcReg1),
                    virtualStack.slotOpConst(srcSlot2, i2));
  // das Ergebnisregister dem Ziel-Slot zuweisen
virtualStack.assignRegisterTo(dstSlot, srcReg1);
}
```

Um Slots zu kopieren wird in dieser Implementierung für den ersten Slot ein Register allokiert, das dann dem zweiten Slot zugewiesen wird. Wenn der erste Slot schon ein Register hatte, so kann das Kopieren übersetzt werden, ohne Code zu erzeugen. Es handelt sich daher um einen sehr einfachen aber suboptimalen Copypropagationalgorithmus.

6.8.2.2 Die eigentliche Registerallokation

Wie oben beschrieben wird ein Register immer dann allokiert, wenn es für den Operanden eines Befehls benötigt wird.

Wenn keine Register mehr zur Allokation zur Verfügung stehen, so wird ein belegtes Register in den Speicher zurückgeschrieben. Die Auswahl des belegten Register erfolgt mit einer sehr einfachen Strategie: Es wird der Nachfolger des zuletzt allokierten Registers genommen. Andere Strategien können durch die Neuimplementierung einer einzelnen Methode installiert werden.

Allerdings ist dabei zu beachten, daß kein Register zurückgeschrieben werden darf, das für den aktuellen Maschinenbefehl benötigt wird. Um das zu vermeiden gibt es in dieser Implementierung einen einfachen Mechanismus: Wenn ein Register allokiert wird, so wird ihm eine 'Generationsnummer' zugeordnet. Beim Allokieren weiterer Register werden nun keine Register befreit, die diese Generationsnummer haben. Für den nächsten Befehl wird nun eine neue Nummer verwendet, was bewirkt daß die zuvor allokierten Register nun wieder befreit werden dürfen.

6.8.3 Die Binärcodeerzeugung

Wie in Kapitel 5.2.9 beschrieben ist, kann die Binärcodeerzeugung bei JIT-Compilern sehr einfach gehalten werden.

In diesem Compiler wird allerdings etwas mehr Aufwand für die Binärcodeerzeugung getrieben. Es gibt extra Klassen, die nur dafür da sind, die Binärdarstellung von Assemblerbefehlen zu erzeugen. Dadurch verbessert sich die Lesbarkeit und Wartbarkeit des Codegenerators des JIT-Compilers. Allerdings wird auch die Laufzeit schlechter.

Im wesentlichen handelt es sich dabei um die Klassen IntInstr, FloatInstr und InstrOperand im Paket 'i386'. Die Klasse IntInstr kann Integerbefehle übersetzen, die Klasse FloatInstr Fließkommabefehle und die Klasse InstrOperand die Operanden von beiden Befehlstypen.

Da diese Klassen von eher geringer Bedeutung sind, soll nur ihre Verwendung an einem einfachen Beispiel erläutert werden: Die Erzeugung des Binärcodes des Maschinenbefehls ADD EAX, [EBP + 10] .

Dazu müssen zuerst zwei Objekte für die Operanden allokiert und initialisiert werden:

```
InstrOperand firstOp = (new
InstrOperand()).register(InstrOperand.EAX);
InstrOperand secondOp = (new
InstrOperand()).memory(InstrOperand.EBP, 10);
```

Danach wird ein Objekt für Integerbefehle initialisiert:

```
IntInstr instr =
(new IntInstr()).setInstruction(IntInstr.ADD, firstOp,
secondOp);
```

Anschließend kann der Binärcode erzeugt werden:

```
int instrPtr = 0;
byte[] code = new byte[100];
instrPtr = instr.translate(code, instrPtr);
```

Das Array 'code' enthält nun die Binärdarstellung des Maschinenbefehls, 'instrPtr' enthält die Zahl der erzeugten Bytes.

Die etwas umständliche Initialisierung der Objekte hat den Zweck, daß die Objekte wiederverwendet werden können, und man nicht für jeden Maschinenbefehl neue Objekte erzeugen muß.

Diese Klassen enthalten außerdem noch Methoden, mit denen sich die Position von Konstanten im compilierten Code ermitteln läßt. Diese werden zum Beispiel für Sprünge benötigt, deren Sprungoffsets während der Codeerzeugung nicht bekannt sind. Der Compiler merkt sich in diesem Fall die Position des Sprungoffsets im Code, und fügt den korrekten Wert an dieser Position ein, sobald dieser bekannt ist.

Leider ist diese Art der Codeerzeugung eigentlich zu langsam für einen JIT-Compiler. Daher gibt es noch eine schnellere Variante. Dabei werden beim Start des Compilers Schablonen für die Binärcodierung von bestimmten Maschinenbefehlen erstellt. In diese Schablonen können dann während der Compilation die aktuellen Register und Konstanten eingetragen werden. Mit diesen Schablonen ist die Geschwindigkeit der Codeerzeugung vergleichbar mit der Variante aus Kapitel 5.2.9. Implementiert sind diese Schablonen mit den Klassen InstrTemplate, FixupImmediate, FixupObject und FixupRegister im Paket i386. Sie werden in der aktuellen Implementierung nur für einige Spezialfälle verwendet.

Die aktuelle Implementierung der Binärcodeerzeugung ist für einen JIT-Compiler viel zu langsam, und muß für einen verwendbaren Compiler neu implementiert werden. Kapitel 7.2 skizziert eine Möglichkeit für eine Neuimplementierung.

6.8.4 Maschinennahe Optimierung

Keine.

Optimierende Compiler beherrschen das sogenannte Instruction Scheduling. Dabei werden die Befehle so angeordnet, daß eine skalarer Prozessor sie optimal verarbeiten kann. Diese Optimierung ist nicht sehr aufwendig, und bringt relativ viel Gewinn. Allerdings würde ihre Implementierung den Codegenerator deutlich komplizierter machen. Sie wurde daher ausgelassen

6.9 Verkettung von Sprüngen

Beim Intel 386 codieren die Sprungbefehle ihr Sprungziel als Differenz zwischen der Anfangsadresse des nächsten Befehls und der Adresse des Sprungziels. Diese Differenz heißt im folgenden Sprungoffset.

Das Problem bei diesen Sprungoffsets ist, daß sie zum Zeitpunkt der Erzeugung nicht bekannt sind, zumindest bei Sprüngen deren Sprungziel noch nicht compiliert wurde. Man muß daher erst die gesamte Methode compilieren, bevor man die Sprungoffsets in den Code eintragen kann. Die Daten für den Sprung müssen solange gespeichert werden. Das sind die Adresse des Sprungoffsets im Code, die Adresse des Befehls nach dem Sprung und die Adresse des Sprungziels.

Für jeden Sprung wird in diesem JIT-Compiler ein spezielles Objekt der Klasse nativecode.UnresolvedJump erstellt. Bei der Erzeugung des Codes für den Sprungbefehl wird in diesem Objekt die Adresse des Sprungoffsets und die Adresse des nächsten Befehls gespeichert. Wenn der Code des Sprungziels erzeugt wird, wird die Adresse des Sprungziels hinzugefügt.

Wenn die gesamte Methode compiliert ist, hat man eine Liste von diesen Objekten, die die gesamten Informationen für das Auflösen der Sprünge enthalten. Diese Objekte können damit selbst die noch fehlenden Sprungoffsets in den Code eintragen.

Das Vorgehen bei Funktionsaufrufen, deren Ziel nicht im compilierten Code liegen ist analog. Hier ist zwar die absolute Adresse des Sprungziels bekannt. Der Intel 386 will bei Funktionsaufrufen aber eine relative Adresse. Dafür braucht man aber die absolute Adresse des compilierten Codes im Speicher, die erst bekannt ist, wenn der Code installiert wird. Daher müssen auch für Funktionsaufrufe Objekte des Typs ‘UnresolvedJump’ verwendet werden.

7 Erweiterungen des Compilers

Dieses Kapitel beschreibt Aspekte, die nicht oder nur teilweise implementiert wurden.

7.1 Inlining

7.1.1 Inlining mit der JVM

Wie bereits in Kapitel 2.3.7 angedeutet, ist das Inlining von Methoden nicht allzu schwer zu implementieren. Man muß nur den Methodenaufruf durch den Code der aufgerufenen Methode ersetzen, und die Position ihres Stackframes anpassen. Die Position des Stackframes der eingebundenen Methode muß dabei so gewählt werden, daß die lokalen Variablen mit den Parametern auf dem Operandenstack überlappen.

Da man bei diesem Compiler zu jedem Befehl ohnehin die explizite Position seiner Operanden speichert, ist das relativ problemlos.

Bei Aufruf nichtstatischer Methoden wird bei der JVM getestet, daß die Referenz auf das Empfängerobjekt nicht Null ist. Dieser Test muß auch bei eingebundenen Methoden beibehalten werden.

Im folgenden Beispiel handelt es sind um Bytecode, bei dem die Positionen der Befehlsoperanden im Stackframe explizit angegeben sind:

Quellcode:

```
; Aufrufer (A)
; a = addInt(a, 2);
 aload 0
 iload 1
 iconst_2
 invokenonvirtual addInt(II)I
 istore 1

; aufgerufene Methode (B)
; int addInt(int x, int y) {return x+y;}
.method addInt(II)I
 iload 1
 iload 2
 iadd
 ret
.end method
```

Die Syntax dieses Codes entspricht dem des Bytecodeassemblers Jasmin [Meyer u.a. 97].

Beim Inlining würden die Stackframes beider Methoden sich folgendermaßen überlappen (l: lokale Variablen, o: Operandenstack):

	Methode A	Methode B
0	A l 0	
1	A l 1	
3	A o 0	B l 0
4	A o 1	B l 1
5	A o 2	B l 2
6		B o 0
7		B o 1

Der erzeugte Code hätte folgenden Aufbau:

```
aload  0 -> 3
iload  1 -> 4
iconst `2' -> 5

checkReference 3   ; Test auf Nullreferenz
iload 4 -> 6
iload 5 -> 7
iadd  6, 7
istore 6 -> 3     ; Ende der Methode B
                  ; ret kopiert Ergebnis in lokale Variable 0
istore 3 -> 1
```

Es handelt sich dabei um bytecodeähnlichen Pseudocode mit expliziten Operanden. Der Code könnte mit einem Copypropagationalgorithmus noch deutlich verbessert werden.

Der Code mit eingebundenen Methodenaufrufen muß nicht unbedingt als Datenstruktur erzeugt werden. Man kann das Inlining während der Codeerzeugung durchführen. Statt Code für einen Methodenaufruf wird dann einfach der Code für den Methodenrumpf erzeugt. Allerdings sind dann weitere Optimierungen nicht möglich.

Das Inlining wurde nur für statische Methoden in einer eher experimentellen Ausführung implementiert.

7.1.2 Inlining von virtuellen Methoden

Wie bereits in Kapitel 5.2.6.2 besprochen, ist das Inlining nur problemlos möglich, wenn zur Compilezeit bekannt ist, welche Methode aufgerufen wird. Bei virtuellen Methoden ist das im allgemeinen nicht der Fall. Allerdings sind virtuelle Methoden in einem Javaprogramm sehr häufig.

Man kann virtuelle Methodenaufrufe einbinden, wenn nur ein mögliches Ziel für den Aufruf zur Compilezeit existiert [Cramer u.a. 97]. Das ist der Fall, wenn die Klasse des Empfängers und ihre Unterklasse die virtuelle Methode exakt einmal definieren. Allerdings muß der compilierte Code dann als ungültig markiert werden, wenn zu einem späteren Zeitpunkt Unterklassen nachgeladen werden, die ein neues mögliches Aufrufziel definieren.

7.2 Schnellere Binärcodeerzeugung

Die aktuelle Implementierung der Binärcodeerzeugung ist für einen JIT-Compiler ungeeignet, da sie zu langsam ist. Die Idee, die Codeerzeugung mit Schablonen zu beschleunigen, führt zu einer deutlich aufwendigeren Implementierung und verschlechtert die Lesbarkeit.

Den Code zur Erzeugung des Binärformats für jeden Befehl von Hand zu erzeugen wie in [Wilkinson 97] erscheint wenig attraktiv, da es einen großen Aufwand bedeutet und der Code schlecht wartbar ist.

Man sollte sich überlegen, ob es nicht möglich ist, den zu erzeugenden Maschinencode in Assemblersprache in den Codegenerator einzubetten. Ein Präprozessor kann dann die Assemblerbefehle aus dem Sourcecode extrahieren und durch Javabefehle zur Codegenerierung ersetzen.

Beispiel:

Der Quellcode des Codes für die Addition eines Slots mit einer Konstante könnte lauten:

```
void generateAddCode(int slot, int constant) {
 int reg = stack.allocRegisterFor(slot);
#ASSEMBLE add REGISTER(reg), IMMEDIATE32(constant)
}
```

Der Präprozessor kann daraus den folgenden Code generieren:

```
void generateAddCode(int slot, int constant) {
 int reg = stack.allocRegisterFor(slot);
 if (doPringDebugMessage)
    System.out.println("add " + regString(reg) + ", " +
      immString(constant));
 code[instrPtr++] = 0x81;
 code[instrPtr++} = 0xc0 | reg;
 instrPtr = insertInteger(constant, instrPtr);
}
```

Die Implementierung des Präprozessors sollte nicht so aufwendig sein, wie es auf den ersten Blick scheint, wenn man einen vorhandenen Assembler zu Hilfe nehmen kann. Der Präprozessor arbeitet dann in zwei Phasen. In der ersten geht er den Quellcode durch, extrahiert die Assemblerbefehle, und erstellt daraus eine Assemblerdatei. Diese wird dann vom Assembler in ein Binärfile übersetzt. Im zweiten Schritt werden die übersetzten Befehle aus dem Binärfile gelesen, und in den Quellcode eingetragen.

Die Schwierigkeit dabei ist, daß der Präprozessor die Positionen der Register und Konstanten in der Binärdarstellung eines jeden einzelnen Befehls herausfinden muß. Dazu läßt er jeden Befehl zweimal mit jeweils unterschiedlichen Registern übersetzen. Wenn er beide Binärdarstellungen exklusiv-oder verknüpft, so kann er die Position der Register einfach erkennen.

Die Technik ist nicht ganz perfekt, da man beim Schreiben des Präprozessors und des Codegenerators immer noch einige Eigentümlichkeiten des Maschinenbefehlssatzes berücksichtigen muß. Außerdem sind Präprozessoren bei moderneren Programmiersprachen wie Java nicht gern gesehen. Soweit ich das sehe ist das dennoch die beste Lösung für das Problem.

Es gibt Assembler, die mehrere Maschinensprachen beherrschen, wie zum Beispiel der GNU-Assembler. Mit einem solchen Assembler sollte es möglich sein, einen Präprozessor zu schreiben, der ebenfalls mehrere Maschinensprachen beherrscht. Zumindest für die erste Phase des Präprozessors sollte ‘Perl’ die optimale Programmiersprache sein.

7.3 Alternativen zur Darstellung des Bytecodes

Wie in Kapitel 6.3 beschrieben, werden bei diesem Compiler alle Bytecodebefehle zuerst in eine Objektdarstellung übersetzt. Das ist relativ ineffizient, denn für die Objekte muß Speicher allokiert werden, und man muß die Objekte initialisieren Wirkliche praktische Vorteile bringt diese Darstellungen nur für fortgeschrittene Anwendungen. Für die reine Übersetzung, wie sie bei JIT-Compilern üblich ist, ist der Nutzen eigentlich relativ gering.

Diese Problem wäre eine typische Anwendung für das Entwurfsmuster ‘Fliegengewicht’ [Gamma u. a. 94]. Dadurch kann man das Allokieren vieler kleiner Objekte vermeiden und trotzdem die gute Aufteilung der Aufgaben auf einzelne Klasse beibehalten. Diese Lösung soll hier nur sehr grob skizziert werden, da ich sie nicht implementiert habe. Man muß dann nur einmal für jeden Bytecodebefehl der JVM ein Objekt erstellen, das dann bei den konkreten Befehlen in einer Methode immer wiederverwendet wird. Bei jeder Operation, die mit einem Objekt ausgeführt wird, wird dem Objekt ein Kontext übergeben, aus dem das Objekt die Informationen erhalten kann, die sonst im Objekt gespeichert wären. Im konkreten Fall würde dieser Kontext z.B. die Parameter des Befehls und den aktuellen Stackpointer enthalten. Die Implementierung wäre etwas aufwendiger als die jetztige Lösung, und würde nicht so gute Erweiterungsmöglichkeiten bieten.

Die meisten JIT-Compiler implementieren die Übersetzung mit Hilfe einer großen Switch-Anweisung, die für jeden Bytecodebefehl eine Routine zur Codeerzeugung hat. Diese Lösung ist schnell, führt aber zu etwas unhandlichen Programmen. Eine Möglichkeit, das vorhandene Programm mit vertretbarem Aufwand zu optimieren, wäre, beide Lösungen miteinander zu verknüpfen: Die aufwendigen Bytecodebefehle werden durch Objekte dargestellt, die einfacheren mit Switch-Blöcken behandelt. Die Klassenhierarchie für die Befehlsobjekte kann man dafür um Methoden erweitern, die den Java Quellcode für die Switch-Anweisungen erzeugen. Dadurch spart man einiges an Schreibarbeit.

Wenn man von der aktuellen Implementierung ausgeht, so läßt sich das Problem der vielen Objektallokationen am einfachsten durch das Wiederverwerten ‘gebrauchter’ Objekte realisieren. Nachdem eine Methode compiliert ist, werden die verwendeten Befehlsobjekte nicht der Freispeicherverwaltung überlassen, sondern in einem Container gespeichert. Für die nächste zu compilierende Methode kann man dann die Objekte aus dem Container verwenden. Der Container besteht aus einem Array, das für jeden Bytecode (Opcode) eine verkettete Liste von Befehlsobjekten für diesen Opcode enthält. Auf diese Weise kann man den Opcodes schnell Objekte zuordnen, und die Objekte können bereits teilweise initialisiert gespeichert werden.

7.4 Speichern von compilierten Code

Da dieser JIT-Compiler recht langsam ist, wäre es attraktiv, wenn man den compilierten Code speichern und bei späteren Aufrufen des Programms wiederverwenden könnte. Das bedeutet, daß der Compiler dann nicht mehr als JIT-Compiler, sondern als NET-Compiler (native executable translation compiler) betrieben wird [Hsieh u.a. 97].

Das Problem dabei ist, daß viele Daten im compilierten Code nach einem Neustart des Programms nicht mehr gültig sind. Bei einem JIT-Compiler kann man alle Konstanten fest in den compilierte Code einfügen. Bei einem NET-Compiler ist das nicht mehr möglich.

Dieses Problem betrifft die folgenden Daten:

- Adressen von Hilfsfunktionen
- Adressen von anderen compilierten Methoden
- Adressen von statischen Feldern
- ID's von Klassen und Methoden
- Positionen von Methoden in der Methodentabelle (je nach JVM)
- Offsets von Feldern von Objekten (je nach JVM)

Wenn man eine compilierte Methode abspeichern will, so muß man zusätzlich zum Code noch eine Tabelle (die Symboltabelle) speichern, die die Daten der Konstanten im Code enthält, die nach einem Neustart neu berechnet werden müssen. Für jede Konstante müssen folgende Daten gespeichert werden:

- die Position der Konstante im Code
- der Typ der Konstante (entsprechend obiger Liste)
- Daten für die Berechnung der Konstante

Die Daten für die Berechnung der Konstante hängen natürlich vom Typ der Konstanten ab. Für die meisten Typen von Konstanten reicht es aus, wenn man den Konstantenpooleintrag, der zur Konstante gehört, kennt. Man muß dann nur die Klasse und den Index des Konstantenpooleintrags speichern. Die Klasse des Konstantenpools mitzuspeichern ist nur notwendig, wenn das Inlining von Methoden erlaubt sein soll.

Die Hilfsfunktionen der JVM lassen sich nicht über Konstantenpooleinträge identifizieren. Hier scheint es das naheliegendste zu sein, jeder Hilfsfunktion eine feste ID zu geben, die dann abgespeichert wird.

Für jeden Typ von Eintrag gibt es eine eigene Klasse. Jeder Eintrag der Symboltabelle ist Instanz einer dieser Klassen. Der Compiler instantiiert und initialisiert Objekte dieser Klassen im maschinenunabhängigen Teil, und übergibt es einer Codeerzeugungsmethode des maschinenabhängigen Teils. Diese kann die Adresse der Konstante im Code eintragen, und das Objekt speichern.

Die Symboltabellen sind noch nicht vollständig implementiert. Die Klasse für die Symboltabelleneinträge ist bytecode.execenv.ConstantPoolSTEntry.

8 Testen des Compilers

Ein wichtiger Schritt bei der Entwicklung des jeden Programms ist das Testen. Für einen Compiler trifft dies in besonderem Maße zu, da Fehler im Compiler bewirken können, daß eigentlich korrekte Programme falsch ausgeführt werden.

Die Testfälle sollte grundsätzlich so gewählt werden, daß jeder Teil des Programms einmal ausgeführt wird, und daß auch seltene und unübliche Eingaben auftauchen. Vorteilhaft ist es, wenn die Tests automatisch durchgeführt werden können. Dadurch kann nach jeder Programmänderung ohne großen Aufwand das gesamte Programm getestet werden.

Diese Aspekte wurden auch beim Test dieses JIT-Compilers beachtet. Dabei wurde für automatische Tests ausgenutzt, daß interpretierter und compilierter Bytecode die gleichen Ergebnisse liefern muß.

Ein einzelner Test läuft daher wie folgt ab: Man hat zwei Methoden mit identischer Signatur und gleichem Rumpf, aber unterschiedlichem Namen. Eine dieser Methoden wird nun compiliert und installiert. Anschließend werden beide Methoden mit jeweils gleichen Parametern aufgerufen, und die Ergebnisse verglichen. Das Ergebnis der interpretierten Methode muß nun identisch mit dem der compilierten Methode sein. Damit Fehler zuverlässig gefunden werden, werden die Methoden normalerweise wiederholt mit verschieden Parametern aufgerufen.

Alle implementierten Testfälle sind in einer Klassenhierarchie organisiert. Jede Klasse implementiert dabei eine Liste ähnlicher Tests.

Zum Beispiel gibt es eine Klasse, die alle arithmetischen Operationen mit Integern testet. Die folgende Klassendeklaration zeigt die Basisklasse dieser Klassenhierarchie (Paket bytecode).

```
abstract class TestBase {
int getNumTest();          // Anzahl der Tests
String getClassName();     // Name der Testklasse
  // Name der für Test `testNumber' zu compilierenden Methode
String getCMethodName(int testNumber);
  // Signatur der für Test `testNumber' zu compilierenden
Methode String getCMethodType(int testNumer);
  // Ausführen von Test `testNumber',
  // die Methode liefert genau dann `true' zurück, wenn der
  // Test erfolgreich war
boolean executeTest(int testNumber);
}
```

Die einzelnen Tests dieser Klasse werden mit von Null beginnenden Indizes identifiziert (testNumber). Für jeden Test kann man über das Interface der Basisklasse den Namen und die Signatur der zu compilierenden Methode ermitteln. Wenn die Methode compiliert und installiert ist, so ruft man TestBase.executeTest() auf. Diese Methode führt den Test durch und teilt dem Aufrufer über den Rückgabewert mit, ob der Test erfolgreich war.

Die Durchführung der Tests erfolgt durch die Klasse bytecode.ControllerTestMeta2.

Da die Methoden einer Testklasse normalerweise sehr große Ähnlichkeit zueinander aufweisen, wurden sie meist automatisch mit Perl-Scripten erzeugt.

9 Bewertung des Compilers

9.1 Entwicklungszustand

Der JIT-Compiler war zum Zeitpunkt der Abgabe der Diplomarbeit noch nicht vollständig. Insbesondere bei der Anbindung an die JVM muß noch einiges implementiert werden.

9.2 Performanz

Da der Compiler noch nicht vollständig implementiert ist, wurde er auch noch nicht richtig optimiert. Insbesondere die Ausführungsgeschwindigkeit muß noch verbessert werden. Auf einem Intel 486 mit 66 MHz kann der Compiler nur circa 75 Bytecodebefehle pro Sekunde übersetzen. Der Compiler selbst wurde dabei interpretiert. Kommerzielle JIT-Compiler erreichen 30000 Bytecodebefehle pro Sekunde, allerdings mit einem schnelleren Rechner (Pentium 100 MHz).

Ausführliche Messungen der Geschwindigkeit des compilierten Codes liegen noch nicht vor. Erste Messung ergeben, daß er in der aktuellen Version eine einzelne Methode etwa viermal so schnell ausführt wie der Interpreter von SUN. Die Geschwindigkeit von compilierten Methoden schwankt aber stark mit der Art der auszuführenden Befehle, so daß diese Zahl nicht allzuviel aussagt. Da er in der aktuellen Version im Prinzip dieselben Techniken verwendet wie andere JIT-Compiler sollte der erzeugte Code auch ungefähr gleich schnell sein.

9.3 Beurteilung des Entwurfs

Das Backend des Compilers ist eindeutig zu langsam. Im nachhinein würde ich hier die Technik mit dem Präprozessor aus Kapitel 7.2 verwenden oder die Binärdarstellung der Befehle von Hand erzeugen. Trotzdem konnte durch die aktuelle Implementierung das Backend relativ einfach, robust und komfortabel realisiert werden.

Die Darstellung der Bytecodebefehle als Objekte führt zu einem guten objektorientierten Design. Allerdings muß das Problem der vielen Objektallokationen noch gelöst werden, am ehesten durch das Wiederverwenden von Objekten (Kapitel 7.3). Dadurch sollte der zusätzliche Aufwand für die Zwischendarstellung durch Objekte vertretbar werden.

Der Compiler war zum Abgabedatum noch nicht vollständig implementiert. Gründe dafür sind einige Fehlentscheidungen beim Entwurf, durch die einige Teile mehrfach implementiert wurden (z.B. nativecode.IMCodeOrig, nativecode.newstack.VirtualStack, stuff.ExecEnvironmentJNI). Außerdem stellte sich die Anbindung an die JVM aufwendiger dar als geplant. Die Optimierungstechniken sind momentan noch in einem eher experimentellen Stadium.

Es ist momentan noch nicht möglich, den Compiler mit sich selbst zu compilieren. Da der Compiler selbst viele kleine Methoden enthält, wäre es interessant, wie gut er sich mit Inlining optimieren läßt.

9.4 Resümee

Im aktuellen Entwicklungszustand ist der Compiler noch zu langsam für den praktischen Einsatz. Im nachhinein betrachtet wäre es sinnvoller gewesen, den Compiler von Anfang an wesentlich einfacher zu halten, und sich mehr auf die Anbindung des Compilers an die JVM zu konzentrieren.

Literaturverzeichnis

Compiler

[Aho u.a. 85] Aho V.Alfred, Sethi Ravi, Ullman D.Jeffrey: Compilers, Principles Techniques and Tools, Addison Wesley, 1985

[Appel 98] Appel W. Andrew: Modern Compiler Implementation in Java, Cambridge University Press, 1998

[Davidson u.a. 84a] Davidson W. Jack, Fraser W. Christopher: Code Selection through Object Code Optimization, ACM Transactions on Programming Languages and Systems Vol.6, No 4, 503-526, October 1984

[Davidson u.a. 84b] Davidson W. Jack, Fraser W. Christopher: Register Allocation and Exhaustive Peephole Optimization, Software-Practice and Experience, Vol 14(9), 857-865, September 1984

[Pittman u.a. 92] Pittman Thomas, Peters James: The Art of Compiler Design, Theory and Practice, Prentice Hall, 1992, New Jersey

[Wirth 96] Wirth Niklaus: Grundlagen und Techniken des Compilerbaus, Addison Wesley, 1996

Intel 386 Assembler

[Abrash 94] Abrash Michael: Zen of Code Optimization; Coriolis Group Books, 1994

[Borland 90] Borland International INC.: Turbo Assembler Referenzhandbuch 2.0., 1990

[Intel 86] Intel Corporation: Intel 80386 Programmer's Reference Manual, 1986

[Intel 95] Intel Corporation: Pentium(R) Processors Family Developer's Manual, Volume 3: Architecture and Programming Manual, 1995

Java

[Agesen u.a. 97] Agesen Ole, Detlefs David: Finding References in Java(tm) Stacks, Sun Microsystem Laboratories, Aug. 1997

[Arnold u.a. 96] Arnold Ken, Gosling James: The Java Programming Language, Addison Wesley, 1996

[Campione u.a. 98] Campione Mary, Walrath Kathy: The Java Tutorial Second Edition, Addison Wesley, 1998

[Dalheimer 97] Dalheimer, Matthias Kalle: Java Virtual Machine: Sprache, Konzept, Architektur; Koeln, O'Reilly, 1997

[Flanagan 97] Flanagan David: Java in a Nutshell, Second Edition, O'Reilly, May 1997

[Lindholm u.a. 96] Lindholm Tim, Yellin Frank: The Java Virtual Machine Specification, Addison Wesley, Reading, Mass., 1996

[Meyer u.a. 97] Meyer Jon, Downing Troy: Java Virtual Machine, O'Reilly, March 1997

[Morrison u.a. 97] Morrison Michael et al.: Java 1.1 Unleashed, Third Edition, Sams.net, 1997

MetaXa

[Golm 97] Golm Michael: Design and Implementation of a Meta Architecture for Java, Institut fuer Mathematische Maschinen und Datenverarbeitung der FAU, Lehrstuhl fuer Informatik IV, 1997

Objektorientierter Entwurf

[Booch 94] Booch Grady: Object Oriented Analysis and Design with Applications, Second Edition, Bejamin/Cummings, Redwood City, CA, 1994

[Fowler 97] Fowler Martin: UML Distilled, Addison Wesley, 1997

[Gamma u.a. 95] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John: Entwurfsmuster, Addison Wesley, 1995

JIT-Compiler

[Chambers 92] Chambers Craig: The Design and Implementation of the SELF Compiler, and Optimizing Compiler for Object-Oriented Programming Languages, Dept. of Computer Science of Stanford University, March 1992

[Cramer u.a. 97] Cramer Timothy et al: Compiling Java Just in Time, IEEE Micro, May/June 1997

[Deutsch u.a. 84] Deutsch Peter., Schiffman Allan: Efficient Implementation of the Smalltalk-80 System, Proc. 11th ACM Symposium on Principles of Programming Languages, pp 297-302, New York, 1984

[Gosling 95] Gosling J.: Java Intermediate Bytecodes, ACM Sigplan Notices Vol 30., No. 3, March 1995

[Grafl 97] Grafl Reinhard: Cacao, Ein 64-Bit-JavaVM-Just-In-Time-Compiler, Technische Universitaet Wien, Institut fuer Computersprachen, 1997

[Griswold 97] Griswold David: Breaking the Speed Barrier: The Future of Java Performance, Java Soft, Sun Microsystems, Inc., Java One 1997, San Francisco, 1997

[Hoelzle u.a. 94] Hoelzle Urs, Ungar David: A Third Generation SELF

Implementation, OOPSLA 1994, Sigplan Notices 1994

[Hsieh u.a. 97] Hsieh A. Cheng-Hsueh et al.: Optimizing NET Compilers for Improved Java Performance, IEEE Computer Vol 30, Iss 6., pp67+, June 1997

Web Sites (zuletzt überprüft am 25.9.98)

[Crelier 97] Interview mit Regis Crelier: Originalseite exitiert nicht mehr - das Interview findet man noch unter http://www.cen.uiuc.edu/~jjones/jit.html

[Mircosoft 98] Just-In-Time Compilation and the Microsoft Win32 VM for Java, http://www.microsoft.com/java/sdk/20/vm/jit.htm

[Wilkinson u.a. 97] Wilkinson Tim et al.: Kaffe Homepage, www.kaffe.org