Universität des Saarlandes
Fachrichtung 6.2 – Informatik

Diplomarbeit

# Realisierung einer Java Virtual Machine mit SEAM

Patrick Cernko

1. September 2004

Angefertigt unter der Leitung von
Prof. Dr. Gert Smolka
und unter der Betreuung durch
Dipl.-Inform. Thorsten Brunklaus

## Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Saarbrücken, den 1. September 2004

Patrick Cernko

# Zusammenfassung

Diese Arbeit beschreibt den Entwurf und die Implementierung von JVM-SEAM. JVM-SEAM ist eine Java Virtual Machine auf Basis des VM-Frameworks SEAM.

Die Entwicklung virtueller Maschinen ist komplex, da viele Dienste für die virtuelle Plattform bereitgestellt werden müssen. Die JVM ist dafür ein gutes Beispiel.

SEAM stellt als Framework zur Entwicklung virtueller Maschinen generische Dienste bereit. Dadurch kann der Entwickler sich mehr auf die spezifischen Dienste und Eigenschaften seiner VM konzentrieren.

In dieser Arbeit wird eine JVM vorgestellt, die SEAM als Grundlage verwendet. Dadurch wird die generelle und konkrete Eignung von SEAM als Basis für beliebige virtuelle Maschinen gezeigt werden.

Die vorgestellte Implementierung integriert Teile einer existierenden JVM in den vorhandenen Prototyp. Durch einen Vergleich der Implementierung mit dem Prototyp und der existierenden JVM soll die Effizienz und Generalität der von SEAM bereitgestellten Dienst gezeigt werden.

## Danksagung

Mein Dank gilt Prof. Dr. Gert Smolka, der mir dieses interessante Thema angeboten und mich in vielen Fragestellungen unterstützt hat.
Insbesondere danke ich Thorsten Brunklaus für seine sehr konstruktive und effiziente Hilfestellung in allen Belangen der Arbeit.
Ich danke den Mitarbeitern des Lehrstuhls und meinen Kommilitonen, die mir für die Beantwortung der vielen Fragen stets bereitwillig zur Verfügung standen.
Nicht zuletzt möchte ich meinen Eltern und meiner Freundin für ihre Unterstützung während meines Studiums danken.

# Inhaltsverzeichnis


**1 Einleitung** 1

1.1 Motivation . . . 1
1.2 Ziel der Arbeit . . . 1
1.3 Aufbau . . . 2

**2 JVM Architektur** 3

2.1 Überblick . . . 3
2.2 Klassenlader . . . 4
2.3 Speicher . . . 6
2.3.1 Datenmodell . . . 6
2.3.2 Code-Speicher . . . 6
2.3.3 Objekt-Speicher . . . 8
2.3.4 Speicherbereinigung . . . 9
2.4 Basisbibliotheken . . . 10
2.5 Ausführung . . . 11
2.5.1 Bytecode-Evaluator . . . 11
2.5.2 Scheduler . . . 12

**3 SEAM** 15

3.1 Überblick . . . 15
3.2 Architektur . . . 15
3.3 Store . . . 17
3.4 Ausführungsmodell . . . 17
3.5 Evaluatoren . . . 19
3.6 Datenmodell . . . 19
3.7 Code-Ausführung . . . 19
3.8 (Un)Pickler . . . 19
3.9 Benutzerdefinierte Dienste . . . 20

**4 Entwurf** 21

4.1 Entwurfsziel und Anforderungen . . . 21
4.2 Architektur . . . 21

4.3 Datenmodell . . . 22
4.3.1 Skalare der Programmiersprache . . . 23
4.3.2 Objekte der Programmiersprache . . . 24
4.3.3 Interne Datenstrukturen . . . 24
4.4 Scheduler . . . 24
4.5 Ausführungsmodell . . . 25
4.6 Speicher . . . 26
4.7 Klassenlader . . . 26
4.8 Basisbibliotheken . . . 27
4.9 Bytecode-Evaluator . . . 27

**5 Implementierung** **31**
5.1 Überblick . . . 31
5.2 Integration des kaffe-Interpreters . . . 31
5.2.1 Abbildung auf die Microsprache . . . 32
5.2.2 Implementierung der Microsprache . . . 33
5.2.3 KaffeInterpreter . . . 34
5.3 Anpassungen zu Integration des kaffe-Interpreters . . . 35
5.4 Probleme bei der Implementierung . . . 35
5.5 Optimierung . . . 36
5.6 Umfang der Implementierung . . . 36

**6 Evaluierung** **39**
6.1 Benchmarks . . . 39
6.2 Ergebnisse . . . 39
6.3 Diskussion . . . 40

**7 Verwandte Arbeiten** **43**
7.1 HotSpot™ . . . 43
7.2 kaffe . . . 43
7.3 SableVM . . . 43
7.4 JikesRVM . . . 44
7.5 Microsoft CLR . . . 45
7.6 Weitere verwandte Arbeiten . . . 45

**8 Zusammenfassung und Ausblick** **47**
8.1 Zusammenfassung . . . 47
8.2 Ausblick . . . 47

**Abbildungsverzeichnis** **49**

**Tabellenverzeichnis** **51**

**Literaturverzeichnis** **53**

# 1 Einleitung

Diese Arbeit beschreibt Entwurf, Implementierung und Evaluierung von JVM-SEAM, einer *Java Virtual Machine* auf Basis des *SEAM*-Frameworks.

## 1.1 Motivation

Die Programmiersprache Java hat sich als eines der erfolgreichsten Programmierkonzepte gezeigt. Einer der wichtigsten Gründe dafür ist die Abstraktion durch die *Java Virtual Machine*.

Eine *virtuelle Maschine* – kurz *VM* – ist ein Programm, das die Dienste und Eigenschaften der definierten virtuellen Plattform auf die zugrunde liegenden konkreten Plattformen abbildet. Die VM erlaubt die Ausführung des spezifischen Programmcodes der Maschine auf der virtuellen Plattform, unabhängig von der konkreten Plattform. Die verschiedenen Dienste und Eigenschaften sorgen für eine relativ hohe Komplexität bei virtuellen Maschinen.

Eine *Java Virtual Machine* ist eine virtuelle Maschine zur Verarbeitung von Java-Bytecode-Programmen, welche die Spezifikation der Java Virtual Machine [15] erfüllt. Durch die exakte Spezifikation erweist sich die Java Virtual Machine als gut geeignete Zielplattform verschiedenster Hochsprachen. Die Entwicklung der Hochsprache beschränkt sich auf die Bereitstellung eines Compilers von der Hochsprache in Java-Bytecode. Die aufwendige Entwicklung einer speziellen virtuellen Maschine entfällt.

An der Universität des Saarlandes wird die *Simple Extensible Abstract Machine* – kurz *SEAM* – entwickelt [7]. SEAM ist ein eine generisches Framework zur Entwicklung virtueller Maschinen. Als Bibliotheksystem bietet es verschiedene Bausteine für VM-Dienste sprachunabhängig an. Grundlage dazu ist ein allgemeines Berechnungsmodell und ein abstrakter Speicher.

Die Verwendung von SEAM erleichtert die Realisierung unterschiedlicher konkreter virtueller Maschinen. Dabei können die von SEAM angebotenen Dienste und Bausteine genutzt, die normalerweise selbst für die virtuelle Maschine bereit gestellt müssen.

## 1.2 Ziel der Arbeit

Ziel der Arbeit ist die Entwicklung einer Java Virtual Machine auf Basis von SEAM. Die Java Virtual Machine stellt dabei eine virtuelle Maschine dar, die sich als Zielplattform für die verschiedensten Hochsprachen gezeigt hat.

Durch die Implementierung einer Java Virtual Machine auf Basis von SEAM soll gezeigt werden, dass die von SEAM angebotenen Dienste genügend abstrakt sind um verschiedene VM-Architekturen zu unterstützen. Andererseits soll gezeigt werden, dass SEAM die Realisierung beliebiger virtueller Maschinen erleichtert.

Durch den Prototyp einer Java Virtual Machine auf Basis von SEAM [7] wurde bereits gezeigt, dass sich SEAM konzeptionell zur Realisierung einer Java Virtual Machine eignet.

Mit dieser Arbeit soll nun gezeigt werden, dass eine *effiziente* Implementierung der Java Virtual Machine mit SEAM möglich ist. Dazu soll der existierende Prototyp verfeinert werden. Anschließend erfolgt eine Evaluierung im Bezug auf Expressivität, Quantität und Qualität der Bausteine von SEAM.

## 1.3 Aufbau

Der Rest dieser Arbeit ist wie folgt aufgebaut. Kapitel 2 beschreibt zunächst die Spezifikation der Java Virtual Machine. Im Anschluss wird in Kapitel 3 SEAM vorgestellt.

Den Entwurf der Java Virtual Machine auf Basis von SEAM beschreibt Kapitel 4. In Kapitel 5 wird die Implementierung des Entwurfs beschrieben. Kapitel 6 zeigt die Evaluierung dieser Implementierung.

In Kapitel 7 werden verwandte Arbeiten vorgestellt. Kapitel 8 fasst die Ergebnisse nochmals zusammen und gibt Ausblicke auf weitere mögliche Arbeiten.

# 2 JVM Architektur

Die definierende Basis einer virtuellen Maschine für Java ist die Spezifikation der *Java Virtual Machine*, kurz JVM [15, 20]. Sie beschreibt, welche Komponenten eine JVM enthalten muss. Ferner wird sehr genau spezifiziert, wie sich die Komponenten im einzelnen und zueinander verhalten müssen. Viele Fakten sind hart festgelegt. Es gibt aber auch Punkte der Spezifikation, in denen nur eine grobe Richtlinie gegeben wird und es im eigenen Ermessen liegt, wie diese Richtlinien konkret implementiert werden. Jedoch existiert immer eine fixierte Vorgabe des Endergebnisses.

## 2.1 Überblick

Abbildung 2.1 zeigt ein Modell der einzelnen Komponenten einer Java Virtual Machine und ihr Zusammenwirken. Es lassen sich die folgenden Komponenten ausmachen:

**Klassenlader** Der Klassenlader hat die Aufgabe die Daten in die Maschine einzulesen. Sie sind in so genannten Class-Dateien gespeichert. Diese enthalten sowohl die Java-Klassen, als auch den zugehörigen Programmcode.

**Speicher** Aufgabe des Speichers ist die Verwaltung aller Datenstrukturen der JVM. Man unterscheidet zwischen der Speicherung von Daten des ausgeführten Programmcodes und dem auszuführenden Codes selbst. Eine wichtige Aufgabe der Speicherverwaltung ist die *automatische Speicherbereinigung*.

**Basisbibliotheken** Aufgabe der Basisbibliotheken ist die Bereitstellung einer standardisierten Grundfunktionalität. Außerdem realisieren sie Schnittstellen zu erweiterten Funktionen, die über die Möglichkeiten des Programmcodes der JVM hinaus gehen. Diese Schnittstellen definieren die *Abstrakte Plattform* der JVM.

**Ausführung** Zur eigentlichen Ausführung des Programmcodes dient der Bytecode-Evaluator. Die Ablaufkontrolle obliegt dem Scheduler, der auch für die korrekte nebenläufige Verarbeitung Sorge trägt.

Im folgenden werden die einzelnen Komponenten detailliert vorgestellt.

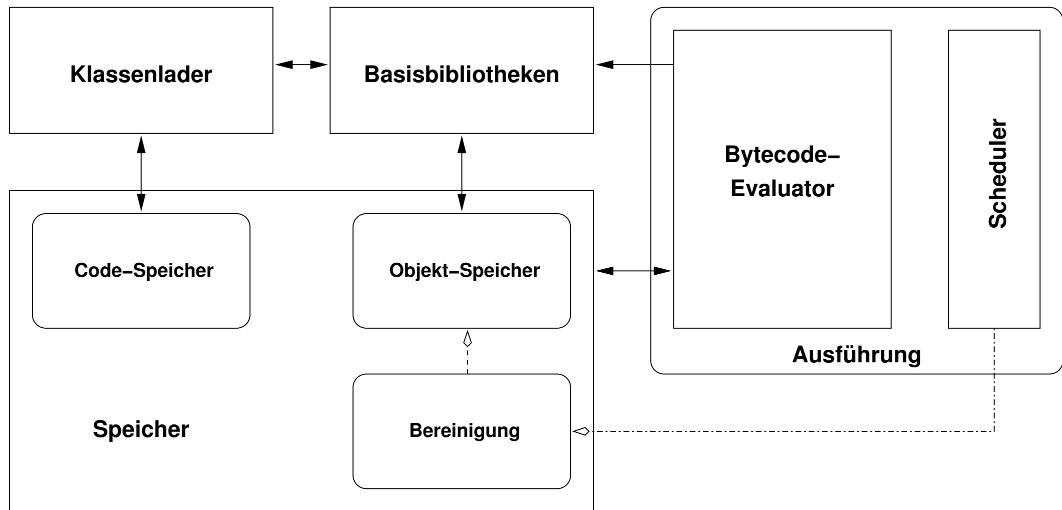


Abbildung 2.1: Die Komponenten einer Java Virtual Machine und ihr Zusammenwirken

## 2.2 Klassenlader

Der Klassenlader (engl. „ClassLoader") ist verantwortlich für das Laden der Java-Klassen. Java-Klassen werden in so genannten *Class-Dateien* gespeichert. Das Format dieser Dateien ist in der Spezifikation der JVM festgelegt.

Eine Class-Datei enthält die Beschreibung exakt einer Java-Klasse. Dies beinhaltet die Beschreibung von:

- abgeleiteter Oberklasse
- implementierten Schnittstellen (engl. „Interface")
- Anzahl, Namen und Typbeschreibungen der statischen und Instanz-Variablen der Klasse (engl. „Members")
- Anzahl, Namen und Signaturen der statischen und Instanz-Methoden

Das Konzept *Interface* zur Spezifikation abstrakter Schnittstellen in der Programmiersprache Java wird in Class-Dateien auf (abstrakte) Java-Klassen abgebildet. Somit existiert keine getrennte Speicherform für Interfaces. Ob es sich bei einer geladenen Java-Klasse um ein Interface handelt ist aber in der Class-Datei vermerkt.

Typen in der Beschreibung der Java-Klasse können entweder andere Java-Klassen sein oder skalare Datentypen der JVM. Andere Java-Klassen werden in der Class-Datei mit einem alphanumerischen Namen bezeichnet. Diese Namen bilden Verweise auf andere Class-Dateien,

welche die entsprechende Java-Klasse beschreiben. Die skalaren Datentypen werden in Abschnitt 2.3.1 noch genauer beschrieben.

Die Class-Datei speichert auch den Programmcode der Methoden der beschriebenen Klasse. Dieser ist in einer plattformunabhängigen Form und wird meist als Bytecode bezeichnet. Er kann also nicht direkt vom Prozessor der ausführenden Plattform verarbeitet werden. Stattdessen muss er in eine Form von der Ausführungseinheit in eine für den jeweiligen Prozessor verwendbare Form gebracht werden. Dies ist Aufgabe des Bytecode-Evaluators, der in Abschnitt 2.5.1 noch genauer vorgestellt wird.

In der einfachsten Form sind diese Daten in Java-Class-Dateien im lokalen Dateisystem der ausführenden Plattform gespeichert. Die Spezifikation der JVM schreibt lediglich diese Speicherart vor. Sie erlaubt allerdings auch, dass die entsprechenden Daten von anderen Quellen wie etwa dem Netzwerk oder Datenbanken stammen können. Hierfür kann der Benutzer eigene abgeleitete Klassenlader als Java-Klassen bereitstellen. Diese haben dann die Aufgabe, die Daten von der entsprechenden Quelle zu laden und dem VM-internen Klassenlader bereitzustellen.

Der Klassenlader hat die Aufgabe, die Beschreibung aus dem einfachen Datenstrom (engl. „Bytestream") einzulesen. Dabei und anschließend muss er einige Verifikationen, welche die Spezifikation vorschreibt, durchführen. Grund für diese Verifikationen ist zum einen natürlich die Vermeidung von Laufzeitfehlern durch fehlerhafte Eingabedaten. Zum anderen dient die Verifikation dem Modell der *Java Security*. Dieses gewährleistet verschiedene Aspekte der Sicherheit in Verbindung mit der Ausführung von Java-Programmen.

**Beispiel** Die Verifikation durch den Klassenlader verhindert beispielsweise, dass eine Klasse der Basisbibliotheken, für den Benutzer ungewollt, von einem ausgeführten Programm durch eine fremde, gefährliche Substitutionsklasse ersetzt wird.

Die eingelesen Daten werden in VM-spezifischen Datenstrukturen im Speicher (vgl. Abschnitt 2.3.2) zur weiteren Verwendung abgelegt. Über die Art und Anforderungen der Datenstrukturen werden in der Spezifikation der JVM keine genauen Vorgaben gemacht.

Der Klassenlader muss auch gewährleisten, dass von einer Class-Datei verwiesene Klassen geladen werden. Dieses Nachladen der Verweise wird mit *Bindung*. Die Spezifikation der JVM macht dabei keine Angaben, wann die Bindung vollzogen wird. Sie definiert stattdessen lediglich, wann sie vollzogen sein muss, wobei abhängig von der Art der Bindung ein früherer oder späterer Zeitpunkt definiert ist. Man erkennt dabei deutlich die Tendenz zu einer möglichst späten Bindung (engl. „lazy binding"). In jedem Fall schreibt die Spezifikation vor, dass ein auftretender Fehler beim Binden erst dann angezeigt werden darf, wenn die Bindung laut Spezifikation vollzogen sein muss. Eine verfrühte Anzeige des Fehlers ist nicht zulässig.

**Beispiel** Eine als Typ einer Klassen-Variable referenzierte Klasse, die nicht gefunden werden kann, darf also erst dann zu einem Programmfehler und Abbruch führen, wenn auf die Variable des entsprechenden Typs zugegriffen wird. Insbesondere muss das Programm ohne diesbezüg-

| Typ | Wertebereich |
|---|---|
| **int** | $\left\{-2^{31}, \ldots, 2^{31}-1\right\}$ |
| **long** | $\left\{-2^{63}, \ldots, 2^{63}-1\right\}$ |
| **float** | $\left\{\pm m \cdot 2^{e-23} \mid m \in \left\{1, \ldots, 2^{23}\right\} \wedge e \in \left\{-(2^7-2), \ldots, 2^7-1\right\}\right\} \cup$ $\{-0.0, +0.0, -\text{INF}, +\text{INF}, \text{NaN}\}$ |
| **double** | $\left\{\pm m \cdot 2^{e-52} \mid m \in \left\{1, \ldots, 2^{52}\right\} \wedge e \in \left\{-(2^{10}-2), \ldots, 2^{10}-1\right\}\right\} \cup$ $\{-0.0, +0.0, -\text{INF}, +\text{INF}, \text{NaN}\}$ |

Tabelle 2.1: Die skalaren Datentypen einer JVM

lichen Fehlerfall terminieren, wenn niemals auf die entsprechende Variable zugegriffen wird.

## 2.3 Speicher

Abbildung 2.1 zeigt den Speicher als Komponente von JVM-SEAM. Man erkennt die Teilkomponenten Speicherbereinigung, Code- und Objekt-Speicher. Der Objekt-Speicher dient zur Ablage von Daten des ausgeführten Programms. Im Code-Speicher wird der Bytecode der geladenen Java-Klassen vom Klassenlader abgelegt. Hauptaufgabe der Speicherbereinigung ist die Freigabe nicht mehr benötigten Speichers.

Die Trennung zwischen beiden Speichereinheiten ist nicht zwingend in der Spezifikation der JVM vorgegeben. Zur einfacheren komponentenübergreifenden Referenzierung und automatischen Bereinigung der gespeicherten Bytecodes ist es laut der Spezifikation sogar möglicherweise sinnvoll beide Komponenten zu vereinigen.

### 2.3.1 Datenmodell

Grundlegend unterscheidet die Spezifikation der JVM zwischen zwei verschiedenen Datentypen: Skalare und Objekte. Tabelle 2.1 zeigt die verschiedenen skalaren Datentypen und deren Wertebereich gemäß der Spezifikation der JVM. Der Wertebereich der Fließkomma-Zahlen stammt aus dem Standard IEEE 754 [1].

Die Speicherung von Objekten ist in Objekt-orientierten Sprachen wie Java extrem wichtig. Die Spezifikation der JVM schreibt jedoch keine Anforderungen für Objekte direkt vor.

### 2.3.2 Code-Speicher

Der Code für den Bytecode-Evaluator wird im Code-Speicher abgelegt. Er muss so gespeichert sein, dass zu jedem Code-Stück die zugehörige Klasse und Methode abrufbar ist. Die genaue Speicherform wird in der Spezifikation der JVM offen gelassen.

```
1 ILOAD_0
2 ILOAD_1
3 IADD
4 ISTORE_0
```

Abbildung 2.2: Die Bytecode-Instruktionen zum Addieren zweier Ganzzahlen

Java-Bytecode, wie er in einer Class-Datei gespeichert ist, sieht wie folgt aus. Zum eigentlichen Code einer Methode werden die folgenden Daten gespeichert:

- die Anzahl der in ihm verwendetet lokalen Variablen[1]
- die Maximalgröße des *Stack* zu seiner Verarbeitung (vgl. Abschnitt 2.5.1)
- eine Tabelle zur Ausnahmebehandlung

Die Tabelle zur Ausnahmebehandlung bildet ein Paar aus Bytecode-Abschnitt und Klassenreferenz auf eine Ausnahmebehandlungsroutine ab. Die Klassenreferenz repräsentiert dabei Ausnahme. Bei der Behandlungsroutine handelt es sich um einen anderen Bytecode-Abschnitt der gleichen Methode, der beim Auftreten der jeweiligen Ausnahme behandelt werden soll.

Der eigentliche Bytecode besteht aus eine Folge von Instruktionen. Jede Instruktion beginnt dabei mit einem eindeutigen 8-bittigen *Mnemonic*. Dieser wird von einer variablen Zahl von Operanden gefolgt. Der Mnemonic kann als ein Befehl der Instruktionsmenge betrachtet werden. Für die meisten Instruktionen ist die Anzahl der Argumente exakt festgelegt, jedoch gibt es auch zwei Ausnahmen. Diese sind die beiden Instruktionen zur Fallunterscheidung. Der Grund dafür ist offensichtlich: Die Anzahl möglicher Fälle ist variabel. Da eine weitere Unterteilung des Bytecodes eine syntaktische und semantische Verarbeitung voraussetzt, wird in den allermeisten Fällen von einer weiteren Unterteilung des Bytecodes bei der strukturierten Speicherung abgesehen.

Die Instruktionen des Bytecodes sind *Stack*-basiert. Das bedeutet, dass für alle möglichen Operanden ein so genannter *Stack* benutzt wird. Er hat die Eigenschaft, dass immer nur sein oberstes Element erreichbar ist. Im deutschen wird eine solche Datenstruktur oft als *Keller* oder *Stapel* bezeichnet. Durch den Stack ist es möglich, dass viele Instruktionen des Bytecodes überhaupt keine Operatoren benötigen.

**Beispiel** Abbildung 2.2 zeigt die Instruktionen zur Addition zweier Ganzzahlen aus den ersten beiden lokalen Variablen einer Methode: Die Instruktionen ILOAD_0 und ILOAD_1 laden den Inhalt der erste beziehungsweise zweite lokalen Variable auf den Stack. Die Instruktion IADD

[1] Zur Vereinfachung werden **long** und **double** Variablen doppelt gezählt da sie ja den doppelten Speicherplatz gegenüber allen anderen Datentypen benötigen.

```
1  public class Object {
2      public Object();
3
4      public native int hashCode();
5      public final native Class getClass();
6
7      public final native void notify();
8      public final native void notifyAll();
9      public final void wait()
10         throws InterruptedException;
11     public final native void wait(long) throws InterruptedException;
12     public final void wait(long, int) throws InterruptedException;
13
14     protected native Object clone() throws CloneNotSupportedException;
15     protected void finalize() throws Throwable;
16     public boolean equals(Object);
17     public String toString();
18 }
```

Abbildung 2.3: Die Schnittstelle der Wurzel-Klasse Object der Programmiersprache Java

nimmt diese beiden Werte nacheinander vom Stack, addiert sie und speichert das Ergebnis wiederum im Stack. Die Instruktion ISTORE_0 nimmt dieses Ergebnis wiederum vom Stack und speichert es in der ersten lokalen Variable.

### 2.3.3 Objekt-Speicher

Wie zu Beginn des Kapitels bereits erwähnt, macht die Spezifikation der JVM keinerlei Vorgaben. Trotzdem ergeben sich aus der Spezifikation bestimmte Eigenschaften, die jedes Objekt erfüllen muss, damit eine effiziente Implementierung möglich ist [5].

Die generellen Eigenschaften, die ein Objekt im Speicher der JVM unterstützen sollte ergeben sich aus den Methoden und Eigenschaften der Klasse Object (vgl. Abbildung 2.3) der Programmiersprache Java, da alle Objekte in der JVM durch Ableitung direkt oder indirekt auf Object basieren.

Jedes Objekt in der JVM muss einen (eindeutigen) Hash-Wert bereitstellen (**int** hashCode()). Ferner muss zu jedem Objekt die zugehörige Klassendefinition anforderbar sein (Class getClass ()).

Nicht direkt ersichtlich ist die Tatsache, dass jedes Objekt gegen konkurrierenden Zugriff gesperrt werden kann (**void** notify (), **void** notifyAll (), **void** wait(), **void** wait(**long**), **void** wait( **long**, **int**)). Das Duplizieren (Object clone()) von Objekten wird in Abschnitt 2.3.4 noch genauer beschrieben.

Das Finalisieren (**void** finalize ()), der Test auf Referenz-Gleichheit (**boolean** equals(Object)) und das Anfordern einer Zeichenkettenrepräsentation (String toString ()) werden in der Klasse Object schon durch Bytecode implementiert.

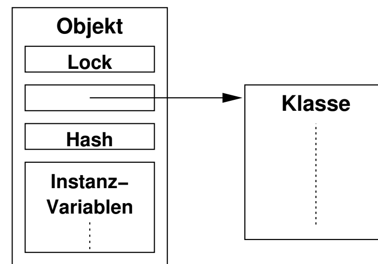


Abbildung 2.4: Der allgemeine Aufbau eines Objekts im Speicher

Nicht aus der Schnittstelle ersichtlich ist, dass die Speicherverwaltung meist Platz für interne Zwecke im Objekt benötigt. Näheres hierzu wird im Abschnitt 2.3.4 beschrieben.

Daraus ergibt sich, dass ein Objekt im Speicher der JVM die folgenden Felder enthalten muss [5]:

**Hash** Im Hash-Feld wird der Hash-Wert des Objektes hinterlegt.

**Class** Jedes Objekt muss einen Verweis auf die Beschreibung seiner zugehörigen Klasse speichern.

**Lock** Zur Speicherung der Objektsperre gegen konkurrierenden Zugriff ist entsprechender Platz im Objekt zu reservieren.

Die meisten Klassen haben Instanz-Variablen die zusätzlich in Instanzen der jeweiligen Klasse gespeichert werden müssen. Aus den von Klassenlader erstellten Datenstrukturen ist ersichtlich, wieviel Speicherplatz dazu im Objekt reserviert werden muss (vgl. Abschnitt 2.2).

Basierend auf diesen Anforderungen visualisiert Abbildung 2.4 den allgemeinen Aufbau eines Objekts im Speicher.

### 2.3.4 Speicherbereinigung

Jeglicher Speicher für Datenstrukturen muss von der Speicherverwaltung angefordert und verwaltet werden. Für den Programmcode des ausgeführten Java-Programms ist eine direkte Anforderung von Speicherbereichen nicht möglich. Um Speicher zu erhalten muss ein entsprechendes Objekt oder einer der skalaren Datentypen verwendet werden.

Außerdem ist es möglich ganze Blöcke eines bestimmten Typs, auch bekannt als Felder (engl. „Arrays”), anzufordern. Bei der Anforderung von solchen Speicherblöcken ist eine Größenangabe notwendig. Felder können nicht vergrößert werden. Stattdessen muss der Programmcode in einem solchen Fall ein neues Feld der neuen gewünschten Größe anfordern und die Daten vom alten Feld in das neue übertragen.

Der angeforderte Speicher wird von der Speicherverwaltung überwacht. Das bedeutet, dass er automatisch wieder freigegeben wird, sobald es kein Objekt in der Speicherverwaltung mehr gibt, das diesen Speicher, dargestellt durch ein Objekt, referenziert. Dieses Verhalten wird als *automatische Speicherbereinigung* (engl. „Garbage collection") bezeichnet [11].

Die Verwaltung und Freigabe von Speicher ist ein relativ komplexer und laufzeitaufwendiger Prozess. Deshalb gilt bei der Speicherverwaltung die Komponente, die sich um die Bereinigung des Speichers kümmert ein besonderes Augenmerk. Der *Garbage Collector* überprüft in regelmäßigen Abständen die Verwaltungsinformationen und gibt nicht mehr referenzierte Speicherbereiche (engl. „Garbage") frei.

Die Spezifikation der JVM macht wenig konkrete Vorgaben zum Garbage Collector, außer dass ein solcher in der Maschine enthalten sein muss. In welchen Intervallen er die Speicherverwaltung prüft wird relativ freigestellt[2]. Auch wird keine Vorgabe darüber gemacht, ob bei einer Speicherbereinigung sämtlicher nicht mehr referenzierter Speicher freigegeben wird.

Eine wichtige Empfehlung der Spezifikation einer JVM ist die Umgruppierung von Speicherblöcken. Da der verwendete Speicher durch die partielle Wieder-Freigabe recht häufig und stark fragmentiert werden kann, ergibt sich bei der Wiederverwendung ein Problem.

Obwohl ausreichend freigegebener Speicher zur Verfügung steht, kann dieser durch seine Partitionierung nicht für große Anfragen verwendet werden. Eine weitere Speicheranforderung der virtuellen Maschine vom Betriebssystem kann aber zu Ressourcenengpässen führen.

Deshalb empfiehlt die Spezifikation der JVM, bei der Speicherbereinigung gleichzeit eine Umgruppierung durchzuführen. Diese Umgruppierung soll noch verwendeten Speicher in freigegebene Speicherplätze verschieben und so für eine Defragmentierung des gesamten verwendeten Speichers sorgen. In der Fachliteratur ist hier meist die Rede vom *Copying Garbage Collector*.

## 2.4 Basisbibliotheken

Die Basisbibliotheken der JVM stellen dem System zunächst eine Grundfunktionalität bereit. Außerdem müssen sie jedoch auch Methoden bereit stellen, die über die Möglichkeiten der Bytecode-Instruktionen hinaus gehen. Dies verlangt meist spezielle Unterstützung durch die virtuelle Maschine.

Beispiele für Anforderungen von spezieller Unterstützung der JVM sind:

- Kommunikation des Java-Programms mit dem Betriebssystem (Ein- & Ausgabeoperationen) und der JVM selbst (*System Properties*)
- Nebenläufigkeit
- *Java Security*

[2] Bei vielen Implementierungen ist es sogar beim Start der Maschine einstellbar.

- *Reflection*
- Unterstützung zum Aufruf von nativen Methoden (engl. „foreign function interface", FFI)

Die von der JVM bereitgestellte für den Basisbibliotheken erforderliche Funktionalität wird mit *Primitive* (engl. „Builtins") *der Abstrakten Plattform* bezeichnet.

## 2.5 Ausführung

Über die Ausführung des Java-Programms wird in der Spezifikation der JVM recht wenig festgelegt. Die meisten Eigenschaften und Aufgaben ergeben sich indirekt.

Der Bytecode des Java-Programms lässt sich in Methoden gruppieren. Daraus resultiert die Anforderung an das Ausführungsmodell, den Bytecode methodenweise abzuarbeiten. Viele Verifikationsfehler dürfen erst bei Verwendung der durch sie fehlerhaften Eigenschaft des Programms angezeigt werden. Daraus resultiert, dass viele Verifikationsschritte erst bei der eigentlichen Ausführung durchgeführt werden sollen.

Im Ausführungsmodell lassen sich aufgrund der Spezifikation der JVM zwei große Komponenten ausmachen: Der Scheduler und der Bytecode-Evaluator. Der Scheduler hat die Aufgabe, die (nebenläufige) Ausführung zu kontrollieren. Die eigentliche Ausführung des Programmcodes geschieht im Bytecode-Evaluator.

### 2.5.1 Bytecode-Evaluator

Die Ausführungseinheit sorgt für die Abarbeitung des Bytecodes. Dazu wird der Komponente genau eine Methode einer vom Klassenlader eingelesenen Java-Klasse übergeben. Zusätzlich erhält die Ausführungseinheit ein eine so genannten Laufzeit-Umgebung, die Adressen der Speicherbereiche für die Parameter und den Rückgabewert der Methode und eine Referenz auf die Java-Klasse der auszuführenden Methode. Die übergebenen Parameter schließen, wie bei objektorientierten Sprachen üblich eine Referenz auf das zugehörige Objekt der Methode mit ein, sofern es sich nicht um eine statische Klassenmethode handelt.

Unter der Laufzeit-Umgebung versteht man einen Speicherbereich zum Ablegen der lokalen Variablen und zur Verwendung als Stack. Durch die Reservierung dieses Speicherplatzes muss der Bytecode-Evaluator während der Ausführung der Methode keine weiteren Speicheranforderungen mehr abwickeln.

Da der Bytecode auf der zugrundeliegenden Plattform nicht direkt ausgeführt werden kann, ergeben sich zwei Wahl-Möglichkeiten zur Ausführung:

**Interpretation** Die Maschine kann den Code gemäß ihrer Semantik abarbeiten. Bei einem solchen Verfahren wird Bytecode-Evaluator meist als Interpreter bezeichnet. Der Interpreter bildet dann die einzelnen Instruktionen auf die Primitive der Implementierungssprache ab.

**Übersetzung in nativen Maschinencode** Die Übersetzung des Java-Bytecodes in nativen Maschinencode vor dessen Ausführung wird allgemein mit **J***ust*-**I***n*-**T***ime (JIT) Compilation* bezeichnet. Dabei wird der Bytecode der auszuführenden Methode vor der eigentlichen Ausführung vollständig in Assembler-Code der ausführenden Plattform übersetzt. Dieser wird dann zur Ausführung gebracht.

Der JIT-Compiler bietet einerseits den Vorteil, dass der assemblierte Code zur Wiederverwendung zwischengespeichert werden kann und die Übersetzung des kompletten Bytecodes der Methode vielfältige Möglichkeiten zur Optimierung des erzeugten Maschinencodes birgt. Andererseits steht dem gegenüber die Zeit, welche zur Übersetzung und Optimierung des Codes benötigt wird. Diese Zeit steht dem Interpreter schon für die eigentliche Evaluierung zur Verfügung.

### 2.5.2 Scheduler

Zur Ausführung von Java-Bytecode wird der Bytecode einer Methode mit einer Ausführungseinheit und einer Laufzeitumgebung in einem so genannten Stack-Frame verknüpft. Unter der Laufzeitumgebung versteht man unter anderem die der Methode übergebenen Argumente, den Rückgabewert und Speicherplatz für zur Methode lokale Variablen.

Ein Thread speichert Stack-Frames in seinem Stack und erlaubt so nur die Ausführung des obersten Stack-Frames. Die Ausführung dieses Stack-Frames kann dazu führen, dass:

- die Umgebung des Stack-Frames modifiziert wird, beispielsweise durch Wertzuweisung einer lokalen Variable oder eines Rückgabewertes.
- der aktuelle Stack-Frame vom Stack entfernt wird (*pop*), was einem Verlassen der Methode entspricht (kontrolliert oder durch Ausnahmewurf)
- Stack-Frames hinzugefügt werden (*push*), was einem Methodenaufruf entspricht

Um Nebenläufigkeit zu ermöglichen können mehrere Threads in der virtuellen Maschine existieren. Die kontrollierende Komponente der Threads ist der Scheduler. Er sorgt für die faire Verteilung der Rechenzeit zwischen ihnen.

Abbildung 2.5 zeigt die modellhafte Arbeitsweise des Schedulers. Er unterbricht in gewissen Zeitabständen die Verarbeitung des gerade aktiven Threads und prüft seine Datenstrukturen. Diese bestehen aus der Referenz auf den aktiven Thread, der Menge der beim Ablauf blockierten Threads und der Warteschlange der zur Ausführung bereiten Threads. Ist der aktive Thread blockiert – beispielsweise durch Ein- oder Ausgabeoperationen – wird er in die Menge der blockierten Threads verschoben, im Bild durch einen gepunkteten Pfeil dargestellt.

Die nicht mehr blockierten Threads werden von der Menge der blockierten in die Warteschlange der ausführbaren verlagert, dargestellt durch einen gestrichelten Pfeil im Bild. War der aktive Thread nicht blockiert, so wird er ebenfalls in die Warteschlange der ausführbaren eingereiht, in der Abbildung wiederum durch einen gestrichelten Pfeil dargestellt. Zuletzt wird der

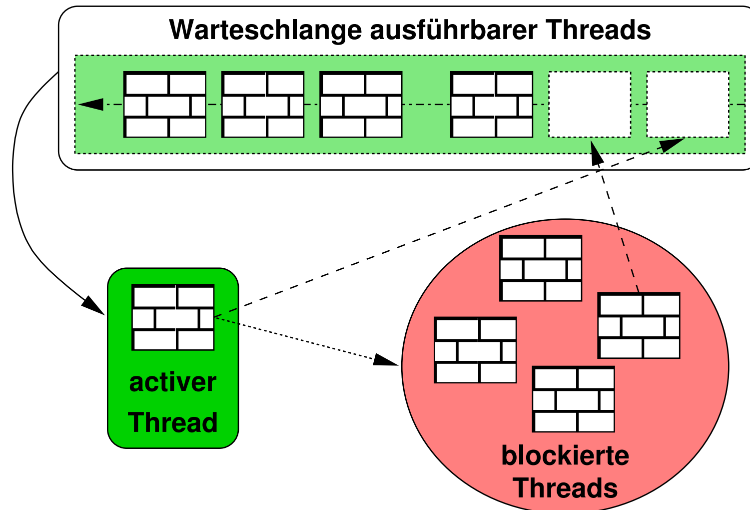


Abbildung 2.5: Die Funktionsweise eines Schedulers

vorderste Thread aus der Warteschlange der ausführbaren als neuer aktiver Thread entnommen, dargestellt durch eine durchgehenden Pfeil. Bis zur nächsten Prüfung wird die Kontrolle dann wieder dem aktiven Thread übergeben.

Die Spezifikation der JVM beschreibt zwei verschiedene Modelle der Nebenläufigkeit. Zum einen ein Modell, das sich des Schedulers und der Möglichkeiten zur Nebenläufigkeit der zugrundeliegenden Plattform bedient. Es bildet einfach jeden Thread der virtuellen Maschine auf einen nativen Thread des Betriebssystems ab. Deshalb wird es in der Spezifikation auch mit *native threads* bezeichnet.

Das zweite Modell bildet Nebenläufigkeit komplett ohne Unterstützung der zugrunde liegenden Plattform. Die Spezifikation der JVM bezeichnet es als *green threads*. Während das erste Modell die Möglichkeiten zur Nebenläufigkeit des Betriebsystems, wie etwa multiple Prozessoren, besser ausnutzt, kann das Modell *green threads* sehr gute Zusicherungen in Bezug auf die Fairness zur parallelen Verarbeitung machen.

# 3 SEAM

Dieses Kapitel beschreibt die Architektur von **S**imple **E**xtensible **A**bstract **M**achine (*SEAM*), einer einfachen und erweiterbaren abstrakten Maschine [7].

## 3.1 Überblick

Ziel des Projekts ist es, die Komponenten und Dienste virtueller Maschinen im allgemeinen zu erforschen. SEAM basiert auf den zwei fundamentalen Erkenntnissen des Projekts:

1. Viele Komponenten einer virtuellen Maschine lassen sich weitestgehend generalisieren und für verschiedenste Maschinentypen verallgemeinern. Diese Komponenten werden im Folgenden als Basiskomponenten bezeichnet.

2. Komponenten, die sich nicht verallgemeinern lassen, können durch allgemeine Schnittstellen repräsentiert werden. Dadurch wird die in SEAM nicht realisierbare Komponente parametrisiert. Im Folgenden werden diese Komponenten als Schnittstellenkomponenten bezeichnet. Eine sprach-spezifische Implementierung einer Schnittstellenkomponente nennen wir Schnittstellenimplementierung.

Die Basiskomponenten von SEAM wurden in einem Kern als Laufzeitumgebung realisiert. Die Validierung dieser Komponenten bezüglich allgemeiner Verwendbarkeit und Effizienz ist ständiges Forschungsthema des Projekts. Über die Schnittstellenkomponenten können sie mit den Schnittstellenimplementierungen kommunizieren und so verschiedene Basisdienste anbieten.

Die Schnittstellenimplementierungen und Dienste einer spezifischen Sprache oder virtuellen Maschine werden in sprach-spezifischen Modulen realisiert, die SEAM als *language layer* bezeichnet, zu deutsch Sprachschicht. Um eine einfache Erweiterbarkeit und Anpassung an eine neue Sprachschicht zu erreichen, wurde eine dynamische Laufzeitumgebung entwickelt. Sie ermöglicht es, die Sprachschichten als *Plugins* dynamisch hinzu zu laden.

## 3.2 Architektur

Abbildung 3.1 veranschaulicht den Aufbau von SEAM. Man erkennt die folgenden Komponenten:

**Store** Der *Store* ist verantwortlich für die Speicherverwaltung.

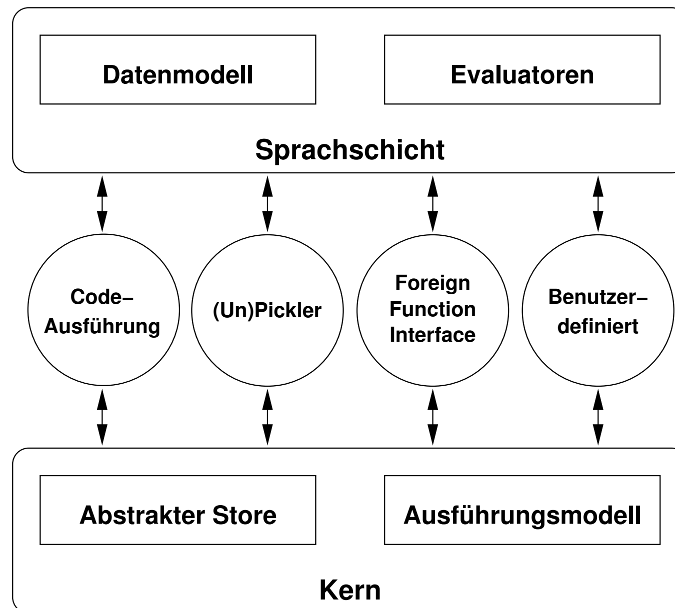


Abbildung 3.1: Die Architektur von SEAM

**Ausführungsmodell** Das Ausführungsmodell realisiert den nebenläufigen Programmablauf.

**Evaluator** *Evaluatoren* sind für die eigentliche Programmverarbeitung verantwortlich.

**Datenmodell** Das Datenmodell der Sprachschicht muss die spezifischen Datenstrukturen auf den abstrakten Store abbilden.

Store und Ausführungsmodell sind als Basiskomponenten im Kern integriert. Datenmodell und Evaluatoren werden durch Schnittstellenkomponenten repräsentiert, für welche die Sprachschicht entsprechende Schnittstellenimplementierungen bereitstellen muss.

Aufbauend auf den Basiskomponenten und unter Verwendung der Schnittstellenkomponenten bietet SEAM die folgenden Dienste:

**Code-Ausführung** Das Ausführungsmodell benutzt die von der Sprachschicht bereitgestellten Code-Evaluatoren zur Ausführung des Programmcodes.

**(Un)Pickler** Die Datenstrukturen des Store sind plattformunabhängig speicherbar und übertragbar.

**Foreign Function Interface** Es existiert eine Schnittstelle zur Einbindung von nativen Funktionen.

**Benutzerdefinierte Dienste** Die Sprachschicht kann eigene Dienste bereitstellen.

Im folgenden werden die einzelnen Komponenten und Dienste des Modells näher beschrieben.

## 3.3 Store

Zur Speicherung der Datenstrukturen dient der Store. Hier werden sowohl die Datenstrukturen des ausgeführten Programms als auch die SEAM-eigenen Daten abgelegt. Es empfiehlt sich außerdem, auch die Datenstrukturen der Sprachschicht im Store abzulegen und so von dessen *automatischer Speicherbereinigung* zu profitieren (vgl. Abschnitt 3.6).

Der Store bietet zur Speicherung verschiedene Datentypen an, nämlich Ganzzahlen bis zu einer Speichergröße von 31 Bit[1], Rohdatenblöcke (*chunks*) und typisierte Blöcke zur Speicherung von Verbunddaten (*blocks*). Darauf aufbauend werden verschiedene komplexere Datenstrukturen, die allgemein gebräuchlich sind angeboten.

Der Store ist auch für die Speicherbereinigung zuständig. Die Verwaltungsinformationen werden bei *chunks* und *blocks* in einer ansonsten nicht sichtbaren zusätzlichen Datenzellen gespeichert. Bei Ganzzahlen muss lediglich eine Markierung gespeichert werden, welche diese als Ganzahlen kennzeichnet. Hierfür wird ein Bit des Datenfeldes verwendet. Das ist auch der Grund, warum dem Ganzzahl-Datentyp in SEAM nur eine 31-bittige Speicherzelle zur Verfügung steht.

## 3.4 Ausführungsmodell

Abbildung 3.2 beschreibt das Ausführungsmodell von SEAM. Es zeigt die folgenden Komponenten:

**Scheduler** Der Scheduler kontrolliert die Ausführung der Threads. Er speichert die Threads in einer Warteschlange, die von hinten aufgefüllt wird. Der jeweils vorderste Thread kommt zur Ausführung. Nebenläufigkeit wird also durch mehrere Threads repräsentiert.

**Thread** Ein Thread besteht aus einem Stapel (engl. „Stack”) von Aufgaben, im Folgenden mit dem englischen Begriff „Task” bezeichnet. Ein Thread delegiert die Ausführung an den Evaluator seines obersten Tasks.

**Task** Ein Task enthält idealisiert eine so genannte *Closure*. Außerdem bietet er Platz für benutzerspezifische Daten.

**Closure** Eine Closure verweist auf den Code, welcher wiederum an den zugehörigen Evaluator verweist. Ferner kann sie benutzerspezifische Daten der Sprachschicht enthalten.

[1] auf 32Bit Prozessor-Architekturen. Derzeit ist eine Portierung von SEAM auf 64Bit-Architekturen geplant, aber noch nicht realisiert. Eine solche Portierung würde die Maximalgröße der Ganzzahl-Speicherzellen vergrößern.

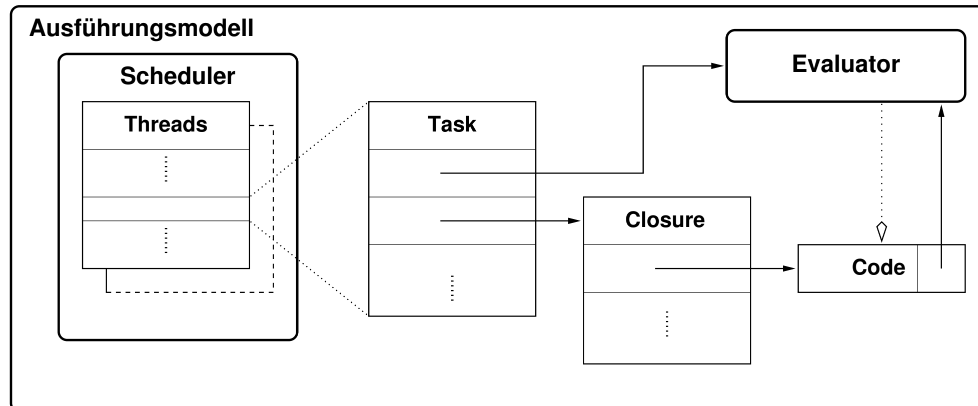


Abbildung 3.2: Das idealisierte Ausführungsmodell von SEAM

**Evaluator** Der Evaluator letztlich verarbeitet den Code, dargestellt durch den gepunkteten Pfeil. Die Verarbeitung wird von der Closure angestoßen. Dabei wird der Code zur Verarbeitung übergeben.

Zur Vereinfachung des Zugriffs enthält der Task ebenfalls eine Referenz auf den Evaluator. Der Evaluator ist von der Sprachschicht bereitzustellen.

Die Ausführung erfolgt folgendermaßen:

1. Der Scheduler nimmt den vordersten Thread zur Verarbeitung.
2. Er übergibt die Kontrolle dem referenzierten Evaluator des obersten Tasks des Threads.
3. Der Evaluator verarbeitet den im Task indirekt referenzierten Code. Bei der Rückgabe der Kontrolle an den Scheduler übergibt er diesem eine Nachricht.

   Der Evaluator kann während der Ausführung den Stack des Threads modifizieren. Er kann:

   - durch die Code-Verarbeitung die Daten der Frames verändern (z.B. Zuweisung an eine lokale Variable)
   - neue Tasks (oberhalb des eigenen) hinzufügen (*push*, z.B. beim Aufruf einer neuen Methode)
   - den aktuellen Task entfernen (*pop*, z.B. nach vollständiger Abarbeitung einer Methode oder abnormer Beendigung der Abarbeitung durch Ausnahmewurf)
4. Der Scheduler übergibt gegebenenfalls der Speicherbereinigung die Kontrolle und erhält sie danach wieder zurück.

5. Der Scheduler prüft die vom Evaluator übergebene Nachricht. Falls keine Präemptionsbedingung vorliegt, beginnt er wieder mit Schritt 2. Andernfalls nimmt er den vordersten Thread aus der Warteschlange. Falls der Thread nicht blockiert ist, wird er am Ende der Warteschlange eingereiht. Danach beginnt der Scheduler mit dem neuen vordersten Thread wieder bei Schritt 1.

Threads, die blockiert[2] sind, speichert der Scheduler in einer Menge blockierter Threads. Sobald sich der Block löst, wird der Thread automatisch wieder in die Warteschlange der Threads des Schedulers eingereiht.

## 3.5 Evaluatoren

Zur Ausführung des Programmcodes wird in SEAM lediglich ein *Container* angeboten. Von der Sprachschicht bereitzustellende Evaluatoren haben über diese Container Zugriff auf den Code. Aufgabe der Evaluatoren ist die Code-Abarbeitung. Oft werden sie deshalb in SEAM auch als *worker* bezeichnet.

Evaluatoren sind nicht zwingend auf die Code-Ausführung festgelegt. Sie können auch zur Modellierung benutzerdefinierter Dienste verwendet werden.

## 3.6 Datenmodell

Die Datenstrukturen der Sprachschicht sollten auf den Store von SEAM abgebildet werden. Grund dafür ist die von Store bereitgestellte *automatische Speicherverwaltung*. Das Datenmodell der Sprachschicht definiert diese Abbildung. Hierzu kann es die vom Store angebotenen Primitive und Datenstrukturen nutzen um die eigenen Datenstrukturen zu modellieren.

## 3.7 Code-Ausführung

Die Ausführung des Programmcodes ist der wichtigste Dienst, den SEAM bereit stellt. Durch das Ausführungsmodell des Kerns wird der Code dabei unter Einbezug der entsprechenden Evaluatoren der Sprachschicht ausgeführt.

Abschnitt 3.4 hat die Funktionsweise dieses Dienstes schon vorweggenommen.

## 3.8 (Un)Pickler

*Pickling* dient dazu, Daten (und damit auch Programmcode) aus der internen Darstellung des Store in eine externe Darstellung zu überführen. Diese externe Darstellung wird *Pickle* genannt.

[2] Ursache für das Blockieren von Threads ist meist das Warten auf das Ein- und Ausgabesystem oder auf andere Threads bei nebenläufigen Programmen.

Mit Hilfe dieser Darstellung ist es möglich die Daten plattformübergreifend zu speichern und zu übertragen.

Das Gegenstück zum *Pickling* ist *Unpickling*. Aus den gespeicherten oder empfangenen Daten einer möglicherweise anderen Plattform werden beim *Unpickling* wieder die Datenstrukturen innerhalb des Store hergestellt.

Durch Pickling und Pickle ist es möglich, den kompletten Zustand der virtuellen Maschine einzufrieren und zu speichern[3]. Außerdem wird die Kommunikation von verteilten Anwendung stark vereinfacht, da der Programmierer sich praktisch nicht um die Übertragung seiner Datenstrukturen kümmern muss.

Vergleichbare Dienste gibt es auch in anderen Systemen, wie beispielsweise *Serialisierung* in Java [16] oder *Corba* zur Programmierung verteilter Anwendungen.

## 3.9 Benutzerdefinierte Dienste

Oft ist es notwendig, für eine bestimmte virtuelle Maschine zusätzliche Dienste zu modellieren. SEAM bietet auch dafür Unterstützung.

Die Datenstrukturen des Dienstes lassen sich auf den Store abbilden. Somit ist eine Interoperabilität mit dem Datenmodell herstellbar. Die *automatische Speicherbereinigung* lässt sich so für den neuen Dienst ebenfalls nutzen, ebenso wie bei Bedarf das *Pickling* und *Unpickling*.

Das Ausführungsmodell lässt sich zur Abarbeitung des Dienstes nutzen. Dazu muss die Sprachschicht lediglich entsprechende Evaluatoren bereitstellen.

[3] Eine Ausnahme bilden natürlich externe Ressourcen wie geöffnete Dateien und Netzwerkverbindungen.

# 4 Entwurf

In diesem Kapitel werden die Modelle und Konzepte vorgestellt, auf denen JVM-SEAM aufbaut. Dabei wird gezeigt, wie SEAM zur Modellierung einer JVM verwendet werden kann.

## 4.1 Entwurfsziel und Anforderungen

Ziel des Entwurfs ist die Implementierung einer lauffähigen Java Virtual Machine. Die Maschine soll dabei folgende Anforderungen erfüllen:

**Benutzbarkeit von SEAM** Die Maschine soll zeigen, dass es möglich und effizient ist, SEAM als Basis für eine JVM zu nutzen.

**Effizienz** Durch die Verwendung von Komponenten aus einer anderen JVM soll ein Leistungsvergleich des Gesamtmodells gemessen werden. Verglichen werden soll dabei die Leistung der neuen Maschine mit der anderen JVM, einer Referenzmaschine und dem bisherigen Prototyp.

## 4.2 Architektur

Als Grundlage für JVM-SEAM dient der existierende Prototyp. Er wurde am Lehrstuhl für Programmiersysteme entwickelt [7]. Das Ziel war eine grundlegende Realisierung einer JVM auf Basis von SEAM. Mit ihr wurde der Beweis erbracht, dass sich SEAM als Grundlage für eine JVM eignet.

Hinzu kommt die virtuelle Maschine kaffe [12]. kaffe ist eine virtuelle Maschine zum Ausführen von Java-Programmen. Als Open-Source Software ist sie frei verfügbar und eignet sich deshalb gut zu Forschungszwecken. Die weiteren Gründe zur Wahl von kaffe werden in den detaillierten Beschreibungen der einzelnen Konzepte von JVM-SEAM aufgeführt.

kaffe stellt seinen Interpreter als Ausführungseinheit für JVM-SEAM bereit. Die saubere und nahtlose Integration dieses Interpreters in das Design von JVM-SEAM war oberstes Ziel.

Das Modell von JVM-SEAM entspricht den in Kapitel 2 vorgestellten Modell der JVM. Konventionelle JVMs müssen alle Dienste und Komponenten selbst bereitstellen. Demgegenüber hat JVM-SEAM den Vorteil, verschiedene Dienste von SEAM nutzen zu können. Eine eigene Implementierung dieser Dienste kann ganz oder zumindest teilweise entfallen.

Abbildung 4.1 zeigt die Architektur von JVM-SEAM. Man erkennt das Modell der JVM als Basis. Komponenten, die durch SEAM bereitgestellt werden sind grün gefärbt. Man erkennt

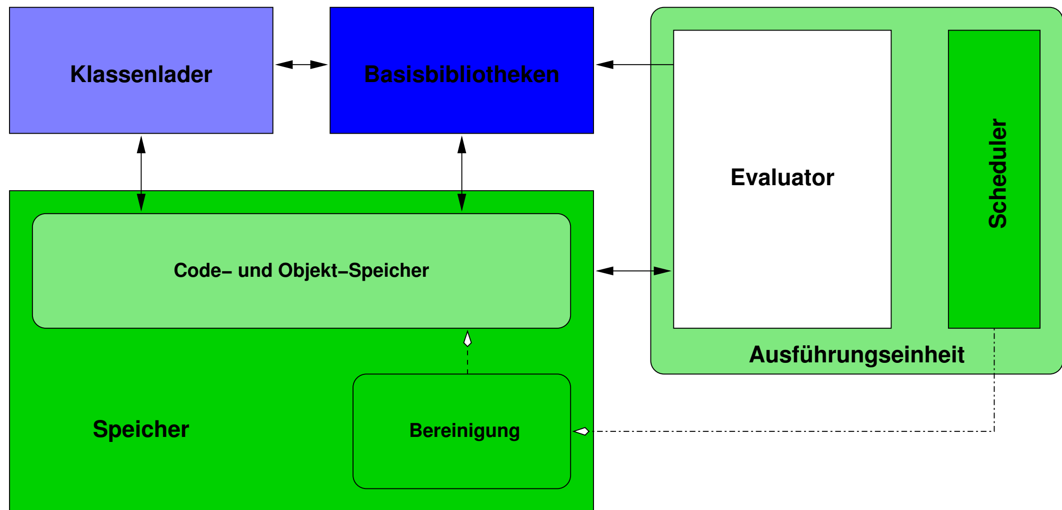


Abbildung 4.1: Die Komponenten der JVM-SEAM

einerseits den Speicher, der auf den Store von SEAM abgebildet wird. Zum Anderen gibt es den neuen Block *Ausführungseinheit*, der das Ausführungsmodell von SEAM mit Scheduler und in diesem Fall einem Evaluator bildet.

Der Objekt- und Code-Speicher werden in JVM-SEAM nicht mehr getrennt. Auch er wird teilweise von SEAM bereitgestellt. Das spezifische Datenmodell zur Abbildung der Datenstrukturen von JVM-SEAM auf den Store von SEAM muss aber bereitgestellt werden.

Die blau gefärbten Komponenten wurden aus dem existierenden Prototyp entnommen. Die Basisbibliotheken fanden dabei vollständig Verwendung. Der Klassenlader musste leicht angepasst werden.

Der in der Ausführungseinheit erkennbare Evaluator ist eine komplette Neuentwicklung. Er basiert auf dem zu Anfang des Kapitels erwähnten Interpreter der virtuellen Maschine kaffe.

## 4.3 Datenmodell

JVM-SEAM bildet die eigenen Datenstrukturen, wie sie im Code- und Objekt-Speicher verwendet werden, auf die Strukturen des Store von SEAM ab. SEAM schreibt keine solche Abbildung zwingend vor. Jedoch liegt es im Interesse des Programmierers, die vom Store angebotenen Datenstrukturen zu nutzen und sämtliche Daten seiner Sprachschicht über eine Abbildung im Store zu speichern.

| Datentyp | Abbildung auf Store |
|---|---|
| **int** | skalares Store-`word`<br>(oder JavaInt als Store-chunk) |
| **float** | Store-float |
| **long** | JavaLong als Store-chunk |
| **double** | Store-double |

Tabelle 4.1: Abbildung der skalaren Datentypen auf Datentypen des Store

Grund für die Abbildung des Datenmodells auf den Store ist die von ihm bereit gestellte Speicherverwaltung. Sie kann nur auf den Daten des Store vollautomatisch arbeiten[1]. Eine eigene Speicherverwaltung der Sprachschicht würde auch ein eigenes Modell zur Speicherbereinigung implizieren. Das komplette Modell von JVM-SEAM würde dadurch weitaus komplexer.

Die Abbildung der Datenstrukturen von JVM-SEAM lässt sich in drei Teilbereiche aufteilen:

- Skalare der Programmiersprache
- Objekte der Programmiersprache
- interne Datenstrukturen von JVM-SEAM

### 4.3.1 Skalare der Programmiersprache

Tabelle 4.1 zeigt die Abbildung der skalaren Datentypen auf den Store. Die Fließkomma-Typen **float** und **double** können direkt auf die entsprechenden Datentypen des Store abgebildet werden. Der Store implementiert diese wiederum als Abbildung auf chunks. Für den Datentyp **long** muss die Sprachschicht selbst eine Abbildung auf einen chunk im Store bereitstellen.

Für den einfachen Ganzzahl-Typ existieren zwei verschiedene Abbildungsmodelle. Das Standardmodell bildet ein **int** auf ein `word` im Store ab, den einzigen skalaren Datentyp, den dieser bietet. Da die Speicherverwaltung ein Bit jedes `word` zur Speicherbereinigung reserviert, ist die Abbildung jedoch unvollständig. Die Spezifikation der JVM verlangt eine Speichergröße von 32 Bit für den Datentyp **int**, nach Abzug des einen Bits für die Speicherverwaltung bietet ein `word` allerdings nur noch eine Größe von 31 Bit an[2].

Die alternative Abbildung des Ganzzahldatentyps erfolgt analog zu **long**: Die Sprachschicht stellt eine Abbildung auf einen chunk im Store bereit. Diese Abbildung führt durch das entstehende *Boxing* zu erheblichen Leistungseinbußen. Deshalb ist sie nur als alternativ enthalten, um eine spezifikationsgemäße Implementierung zu ermöglichen.

[1] Verwendet man im Store externe Referenzen – z.B. Datei-Handler – so muss man zusätzlichen Code bereitstellen, der vom Store zur Finalisierung verwendet wird.

[2] Eine mögliche Portierung von SEAM auf eine 64Bit-Architektur würde diese Einschränkung dort aufheben.

### 4.3.2 Objekte der Programmiersprache

Für die Speicherung von Objekten muss die Sprachschicht eine Abbildung auf den Store bereitstellen. Zum Einen müssen die in Abschnitt 2.3.1 beschriebenen Feldern für Lock, Hash und Klassenreferenz gespeichert werden. Für jedes Feld der korrespondierenden Java-Klasse muss der entsprechende Speicherplatz reserviert werden.

Die Abbildung dieser Felder basiert auf dem Modell des Prototyps. Jeder Speicherplatz wird auf einen Speicherplatz des Store abgebildet. Das Objektmodell basiert daher auf einem Block des Store. Jedes Feld des Blocks entspricht dabei einem Feld des Objekts. Für die allgemeinen Felder werden spezialisierte Schnittstellen zu deren Abfrage und Setzen im Modell bereitgestellt. Die Typ-spezifischen Felder werden über Index-basierte Schnittstellen verarbeitet.

### 4.3.3 Interne Datenstrukturen

Um die Speicherverwaltung von SEAM ohne eigene Erweiterungen nutzen zu können werden auch die internen Datenstrukturen von JVM-SEAM auf dem Store abgelegt. Darunter fallen die verschiedenen Datenstrukturen des Klassenladers. In ihnen werden die verarbeiteten Daten der eingelesenen Java-Klassen gespeichert. Die Datenstrukturen des Ausführungsmodells – verschiedene spezialisierte Stack-Frames – werden ebenfalls auf den Store abgebildet. Die zur Komponente der Basisbibliotheken zählenden nativen Methoden werden als Store-basierte Tabelle referenziert.

## 4.4 Scheduler

Die Steuerung des Programmablaufs erfolgt durch den SEAM-Scheduler. Er genügt den Anforderungen von JVM-SEAM hinreichend. Lediglich die in der Spezifikation der JVM beschriebenen *native threads* werden von SEAM derzeit noch nicht unterstützt. Die Unterstützung letzterer ist allerdings in der Spezifikation der JVM als optional deklariert.

Der Scheduler verwaltet – wie in Abschnitt 2.5.2 beschrieben – eine Warteschlange von Threads. Aus dieser Warteschlange nimmt der Scheduler den jeweils vordersten Thread[3] . Seine Ausführung bewirkt die Ausführung seines obersten Stack-Frames. Diese erfolgt, bis zum Erreichen der Präemptionsbedingung oder der Blockierung der weiteren Ausführung. Der Scheduler übernimmt den Thread wieder und ordnet ihn je nach Zustand in die Warteschlange oder die Menge der blockierten Threads ein.

[3] Im allgemeinen würde ein Scheduler ebensoviele Threads entnehmen, wie Prozessoren zur Verfügung stehen. SEAM ist aber derzeit noch auf einen Prozessor beschränkt.

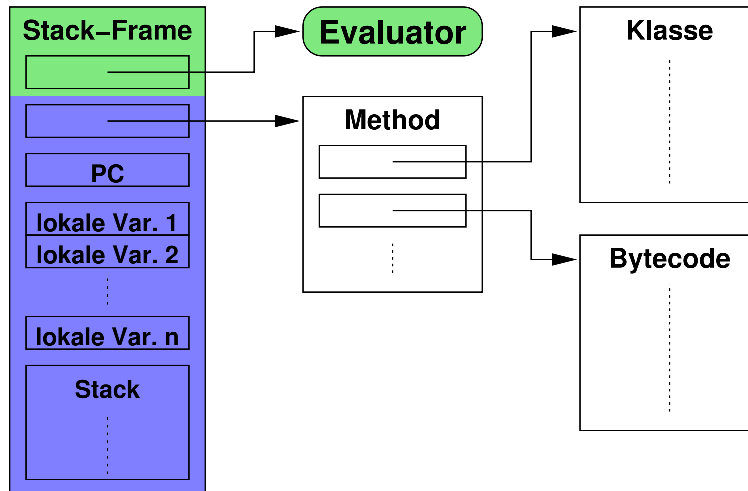


Abbildung 4.2: Der Aufbau eines Stack-Frames

## 4.5 Ausführungsmodell

Als Ausführungsmodell von JVM-SEAM wird das Modell von SEAM benutzt. Das Modell von SEAM bietet alle von der JVM geforderten Eigenschaften. Es bietet direkt Nebenläufigkeit. Die Stack-Frames der JVM-Spezifikation lassen sich direkt auf Tasks des SEAM-Modells abbilden.

Ein Stack-Frame von JVM-SEAM ist wie in Abbildung 4.2 dargestellt aufgebaut. Zunächst enthält es einen Verweis auf den Evaluator. Im Fall von JVM-SEAM wird der Evaluator durch den *KaffeInterpreter* repräsentiert, der in Abschnitt 4.9 genauer vorgestellt wird. Ferner speichert der Stack-Frame eine Referenz auf die Java-Methode, die zur Ausführung gebracht werden soll. Über diese Referenz hat der Evaluator sowohl Zugriff auf den Bytecode der Methode als auch auf die zur Methode gehörende Java-Klasse. Des Weiteren wird Platz für die Speicherung der Arbeitsumgebung (engl. „environment") des Evaluators reserviert.

Die Arbeitsumgebung beinhaltet einen Bereich zur Speicherung der aktuellen Position im zu verarbeitenden Bytecode. Diese Speicherstelle wird im englischen meist mit „program counter" oder kurz *PC* bezeichnet. Daneben besteht die Arbeitsumgebung aus zwei großen Teilbereichen. Den ersten Teil bildet der Speicherplatz für die lokalen Variablen der Methode. Da die Anzahl und der Typ (und damit der Platzbedarf) der lokalen Variablen beim Erstellen eines Stack-Frames bekannt ist, kann dieser Platz sofort bei der Erstellung des Stack-Frames reserviert werden.

Der zweite Teil der Arbeitsumgebung ist für den Variablen-*Stapel* – im Folgenden gemäß der Spezifikation der JVM einfach *Stack* genannt – reserviert (vgl. Abschnitt 2.5.1). Auch seine Maximalgröße ist über die auszuführende Methode bei der Erstellung des Stack-Frames bekannt und kann sofort reserviert werden.

Man erkennt den für einen Task von SEAM notwendigen Teil eines Stack-Frames in der Abbildung 4.2 in grün. Die blau unterlegten Teile sind für SEAM-Task benutzerspezifische Daten.

Neben dem Bytecode-Evaluator zur Ausführung des Programmcodes gibt es noch einen Evaluator für den Klassenlader. Die beiden Evaluatoren werden in Abschnitt 4.9 und 4.7 vorgestellt.

## 4.6 Speicher

Für die Speicherverwaltung und -bereinigung von JVM-SEAM wird eine Abbildung auf den Store von SEAM verwendet. Grund dafür ist zum Einen die effiziente Implementierung in SEAM. Zum Anderen ist eine Speicherverwaltung und vor allem die Speicherbereinigung relativ komplex. Außerdem liegt eines der Ziele der Arbeit in der Überprüfung der Verwendbarkeit von SEAM für JVM-SEAM. Eine eigene Implementierung der Speicherverwaltung würde der Studie den Grund entziehen.

Somit muss in der JVM-SEAM lediglich die Abbildung der eigenen Datenstrukturen auf den Store modelliert werden. Dies wurde bereits in Abschnitt 4.3 beschrieben.

## 4.7 Klassenlader

Der Klassenlader von JVM-SEAM wurde größtenteils vom Prototyp übernommen. Lediglich bei der Speicherung der statischen und Instanz-Methoden wurden für den neuen Bytecode-Evaluator einige Verfeinerungen vorgenommen. Die ursprüngliche Version benötigte den Zugriff auf Namen und Signatur der Methode nicht.

Zum Laden und Verifizieren einer Klasse benutzt der Klassenlader das Ausführungsmodell von SEAM. Er stellt dazu einen Evaluator bereit. Der Evaluator lädt die Klasse, deren Name ihm in einem entsprechenden Stack-Frame übergebenen wurde. Muss also bei der Programmausführung eine neue Java-Klasse geladen werden, so wird dem aktiven Thread ein *BuildClassFrame* angefügt, das dann mit Hilfe des zugehörigen *BuildClassWorkers* evaluiert wird. Dadurch wird die Klasse geladen.

Treten beim Laden keine Fehler auf, so wird nach dem abgeschlossenen Ladevorgang das BuildClassFrame entfernt, worauf der normale Programmablauf mittels des Bytecode-Evaluators fortgesetzt wird. Im Fehlerfall wird eine entsprechende Ausnahme erzeugt, die durch das Ausführungsmodell im Stack des Threads nach oben gereicht wird.

Die Spezifikation der JVM empfiehlt die *späte Bindung* der Referenzen einer zu ladenden Klasse. Deshalb verwendet der Klassenlader von JVM-SEAM *Futures* aus SEAM um noch nicht geladene Referenzen zu realisieren. Bei der ersten Verwendung einer solchen Referenz wird die *Future* aufgelöst, indem die hinterlegte *Closure* die Bindung der entsprechenden Referenz mittels des Evaluators des Klassenladers bewirkt. Dadurch entsteht das von der Spezifikation der JVM empfohlene *lazy binding*.

## 4.8 Basisbibliotheken

**Existierender Prototyp** Der Prototyp verwendet als Basisbibliotheksystem das System der HotSpot™ JVM [19]. Hierzu wurden die notwendigen Teile diese Systems integriert. Eine Anpassung dieser Teile war nicht notwendig. Die benötigten nativen Methoden der Abstrakten Plattform der JVM werden in geradliniger Weise bereitgestellt.

Zur vereinfachten Fehlersuche wurde im Prototyp zusätzlich zu o.g. System ein einfaches Ausgabesystem integriert. Es bietet die Möglichkeit, die verschiedenen Basistypen sowie Zeichenketten auszugeben. Ebenso ist eine generische Ausgabe von Java-Objekten möglich.

**Virtuelle Maschine kaffe** Das Projekt GNU Classpath [10] entwickelt ein freies Basisbibliotheksystem für JVMs. Das System wird in der virtuellen Maschine kaffe [12], der SableVM [9] und der Jikes RVM [2] (vgl. Abschnitt 7.3 und 7.4) verwendet. Dabei muss das System bisher bei allen Projekten leicht auf die jeweiligen Systeme angepasst werden.

Zur Vereinfachung der Entwicklung von JVM-SEAM wurden den bereits im Prototyp hinzugefügten Erweiterungen zur Datenausgabe weitere hinzugefügt.

Eine Übernahme von GNU Classpath – dem Modell der virtuellen Maschine kaffe – wurde nicht in Betracht gezogen. Eine Adaption dieses Modells stellte keine nennenswerte Leistungsverbesserung dar. Weiterhin ist die Schnittstellenmodellierung – besonders die Bereitstellung der benötigten nativen Methoden der *Abstrakten Plattform* – eine langwierige und fehlerprovozierende Arbeit. Sie steht im keinem Verhältnis zum zu erwartenden Leistungsgewinn.

## 4.9 Bytecode-Evaluator

Zur Modellierung des neu entwickelten Interpreters von JVM-SEAM wurde der Interpreter der virtuellen Maschine kaffe [12] als Vorlage benutzt.

kaffe bietet verschiedene Modelle von Ausführungseinheiten. Zum einen gibt es einen Interpreter. Daneben existieren derzeit drei verschiedene JIT-Compiler-Modelle (vgl. Abschnitt 2.5.1). Bisher sind diese nicht gleichzeitig in der selben Maschine verwendbar.

Alle vier Ausführungseinheiten basieren auf einer gemeinsamen Abstraktion des Bytecodes in allgemein verwendbare Primitive. Die abstrakten Primitive bilden eine Menge die im folgenden *Microsprache* genannt wird. Die einzelnen Ausführungseinheiten stellen Implementierungen dieser Primitive bereit.

**Beispiel** Abbildung 4.3 zeigt beispielhaft die Abbildung der Instruktion ILOAD – Laden einer Ganzzahl aus einer lokalen Variable – auf die Primitive der Microsprache von kaffe. Zunächst erfolgt eine Debug-Ausgabe des aktuellen Wertes des *program counters*. Danach wird der für die Instruktion notwendige Parameter anhand der aktuellen Position im Bytecode ausgelesen. Es folgt die Debug-Ausgabe der Instruktion inklusive Parameter und die Debug-Ausgabe der

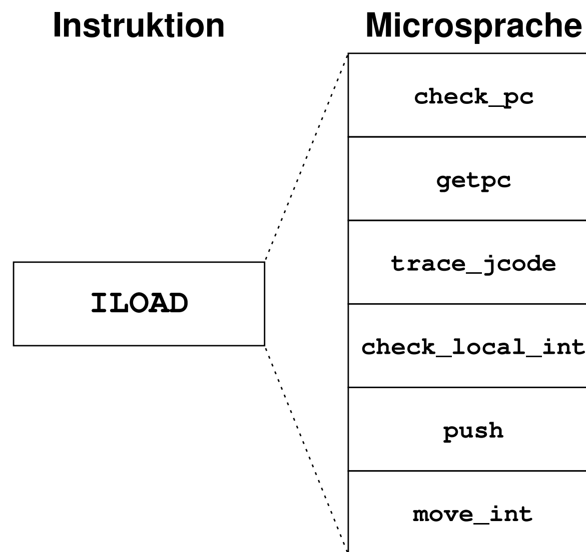


Abbildung 4.3: Die Bytecode-Abstraktion der virtuellen Maschine kaffe

betreffenden lokalen Speicherzelle. Der Variablen-Stack wird um eine Speicherzelle erweitert. Zuletzt wird der Wert der lokalen Variable in die neue Stack-Speicherzelle übertragen.

Durch die Unterteilung in Microsprache und Implementierung eignet sich der Interpreter der virtuellen Maschine kaffe besonders zur Verwendung in JVM-SEAM. Die Integration des Interpreters verwendet die Abstraktion durch die Microsprache, um eine eigene Implementierung bereit zu stellen. Dabei wird die Implementierung möglichst an die Implementierung des Interpreters von kaffe anlehnt. Wo dies nicht möglich war mussten Anpassungen anhand des kaffe-Interpreters als Vorlage gemacht werden. Diese Anpassungen lassen sich in drei Bereiche unterteilen:

**Datenmodell** Da JVM-SEAM ein anderes Datenmodell als kaffe verwendet, mussten für alle Primitiv-Implementierungen der Microsprache, auf das neue Datenmodell Modell angepasst werden. Bei einfachen Primitiven – wie beispielsweise mathematischen Operationen – besteht diese Anpassung meist lediglich aus einer 1 : 1-Abbildung zwischen den beiden Modellen.

Komplexere Mechanismen – wie beispielsweise der Objektzugriff – erfordern jedoch mehr Aufwand. Sie führten oft auch zu den schon erwähnten Erweiterungen des Datenmodells, die eine einfache und effiziente Implementierung des Interpreters erst ermöglichten.

**Abstrakte Plattform** Der Zugriff auf die Abstrakte Plattform von JVM-SEAM musste weitgehend neu modelliert werden. Hier zeigte sich am deutlichsten der Unterschied in den

Modellen von kaffe und JVM-SEAM. Nicht selten war hier nur eine Anpassung möglich, die den dritten Teilbereich mit einschloss.

**Indirekte Anpassung der Microsprache** In einigen Fällen war die Microsprache in ihrer Untergliederung zu fein oder andersartig als dass eine Implementierung der Primitive für JVM-SEAM möglich gewesen wäre. In solchen Fällen musste das Interpretermodell durch die gegenseitige Verschachtelung der Implementierung mehrerer Primitive der Microsprache in gegenseitiger Wechselwirkung angepasst werden. Erst die Einführung solcher gegenseitiger Abhängigkeiten brachte die gewünschte Semantik der jeweiligen Instruktionen.

**Beispiel** Zur Implementierung der Methodenaufrufe verwendet die Microsprache von kaffe zunächst die Instruktion build_call_frame (). Gemäß ihrem Namen wird sie in JVM-SEAM derart modelliert, dass die Implementierung der Instruktion die für den Aufruf einer neuen Methode erforderlichen Datenstrukturen zusammenstellt. Erst die Instruktion call_indirect_method () jedoch darf die eigentliche *Ausführung*, also das Einsetzen eines neuen StackFrames auf den Stack des Threads und die Übergabe der Verarbeitung an dieses StackFrame, anstoßen.

Die Möglichkeiten der Weitergabe der Datenstrukturen von der ersten zur zweiten Instruktion sind begrenzt. Um eine redundante Teilimplementierung für beide Instruktionen zu vermeiden, wird die Strukturierung durch die Instruktionen umgangen und in der zweiten Instruktion das Vorhandensein bestimmter Daten implizit angenommen und zur Implementierung benutzt.[4]

Die Idee zur gegenseitigen abhängigen Verschachtelung der Implementierung von Primitiven der Microsprache stammt teilweise schon aus dem kaffe-Interpreter, wurde jedoch stark erweitert. Nichtsdestotrotz wurde sie nur angewendet, wenn es keine andere Möglichkeit der Anpassung gab, ohne noch weiter vom bestehenden kaffe-Modell abzukommen. Sie verleiht dem Modell eine höhere Komplexität, die nicht gewünscht ist.

[4] In der Implementierung von JVM-SEAM spiegelt sich diese Tatsache darin wieder, dass die Expansion der beiden Instruktionen nur gemeinsam einen sinnvolle Semantik ergibt. Die Implementierung der ersten Instruktion würde allein genommen nur einigen *lokalen* Variablen Werte zuweisen, die der zweiten würde nicht deklarierte Variablen verwenden. Durch die Implementierung der Microsprache durch Makroexpansion statt Funktionsapplikation ist diese Technik überhaupt möglich.

# 5 Implementierung

Im vorangegangenen Kapitel wurden die Modelle von JVM-SEAM vorgestellt. Basierend auf diesen Modellen erfolgte die Implementierung. Im Folgenden werden die Kernpunkte und Probleme bei der Entwicklung von JVM-SEAM dargestellt.

## 5.1 Überblick

Der Code von JVM-SEAM basiert auf dem Prototyp, wie er zur Evaluierung von SEAM [7] entwickelt wurde. Einige Teile dieses Prototyps konnten unverändert übernommen werden. Dazu zählt vor allem die Schnittstelle zum Basisbibliotheksystem.

Bei anderen Komponenten, wie Datenmodell und Klassenlader mussten leichte Anpassungen vorgenommen werden. Meist wurden diese Anpassungen durch die neu hinzugekommene Ersatzkomponente impliziert.

Die Hauptarbeit lag in der Integration des Interpreters der virtuellen Maschine kaffe [12].

## 5.2 Integration des kaffe-Interpreters

Die neue Ausführungseinheit von JVM-SEAM basiert auf dem Interpreter der virtuellen Maschine kaffe [12]. Ziel der Integration war die Wiederverwendung möglichst großer Programmcodeteile. Insbesondere die Abstraktion der Microsprache (vgl. Abschnitt 4.9) sollte möglichst unverändert weiterverwendet werden.

Abbildung 5.1 zeigt den Aufbau des Interpreters. Die folgenden Komponenten sind erkennbar:

**KaffeInterpreter** Der KaffeInterpreter dient zur Einbindung in das Ausführungsmodell von SEAM. Er stellt nach außen die Schnittstelle eines Evaluators bereit.

**Abbildung auf die Microsprache** Die Abbildung der Bytecode-Instruktionen auf die Microsprache von kaffe wird durch die Datei `kaffe.def` von der virtuellen Maschine kaffe

Abbildung 5.1: Die Komponenten zur Implementierung des Interpreters

```
100 define_insn(ILOAD)
101 {
102         /*
103          * ..., -> ..., local variable
104          */
105         check_pc (0);
106         idx = (uint8)getpc(0);
107         trace_jcode (("iload %d\n", idx));
108 
109         check_local_int(idx);
110 
111         push(1);
112         move_int(stack(0), local(idx));
113 }
```

Abbildung 5.2: Ein Ausschnitt der Bytecode-Abbildung auf die Microsprache von kaffe

bereitgestellt. Der KaffeInterpreter bindet diese Abbildung in seine Verarbeitungsroutine ein.

**Implementierung der Microsprache** Durch die Implementierung der Microsprache werden die Bytecode-Instruktionen letztlich in Primitive der Programmiersprache C++ umgesetzt (= interpretiert). Die Implementierung orientiert sich an der Implementierung für den Interpreter innerhalb der virtuellen Maschine kaffe.

### 5.2.1 Abbildung auf die Microsprache

Die Abbildung des Java-Bytecodes auf die Microsprache von kaffe erfolgt in der Datei `kaffe.def`. Die Datei dient in kaffe als Basis für die Implementierung des kaffe-Interpreters und der drei JIT-Compiler. Mittels einer kleinen Makro-Sammlung wird jede Bytecode-Instruktion auf die zugehörige Anweisungsfolge der Microsprache abgebildet.

Abbildung 5.2 zeigt einen Ausschnitt von `kaffe.def`. Sie zeigt die Abbildung der Instruktion ILOAD auf den zugehörigen Code der Microsprache.

Da die Microsprache für die Verwendung mit einen C-Compiler ausgelegt ist, mussten leichte syntaktische Anpassungen vorgenommen werden. So wurde beispielsweise eine Variable mit Namen class verwendet. Da der Code in JVM-SEAM mit einem C++-Compiler übersetzt werden sollte, musste solch ein Variablenname durch ein nicht-reserviertes Wort ersetzt werden.

kaffe bildet Felder intern auf Objekte mit bestimmten Eigenschaften ab. Deshalb sieht die Microsprache keine speziellen Instruktionen zum Zugriff auf Felder vor, sondern behandelt sie als Objekte. Da JVM-SEAM eigene Datenstrukturen für Felder bietet, musste die Microsprache beim Zugriff auf Feldeigenschaften und -daten verfeinert werden. Speziell für den Zugriff

auf die Feldlänge wurde dafür die neue Instruktion load_arraylength eingeführt, welche die allgemeineren Objektzugriffe bei der Implementierung der Bytecode-Instruktion ARRAYLENGTH ersetzte.

Bis auf die hier beschriebenen Änderungen wurde die Abbildung auf die kaffe-Microsprache unverändert in JVM-SEAM verwendet.

### 5.2.2 Implementierung der Microsprache

Das zweite Ziel der Arbeit war die Bereitstellung einer Abbildung von der Microsprache von kaffe auf die Primitive der Programmiersprache C++, in der JVM-SEAM implementiert ist. Zusammen mit der Abbildung auf die Microsprache bildet die zweite Abbildung den neuen Interpreter von JVM-SEAM.

Die Implementierung richtet sich nach der entsprechenden Implementierung im Interpreter der virtuellen Maschine kaffe. Sie ist auf die drei Dateien `DefaultKaffeAccessors.hh`, `KaffeAccessors.hh` und `kaffe-soft.c` verteilt. Hinzu kommen die Schnittstellendefinitionen wie sie für die Microsprache notwendig sind.

#### DefaultKaffeAccessors.hh

In `DefaultKaffeAccessors.hh` werden die Implementierungen der Microsprache angegeben, wie sie in der Implementierung des Interpreters von kaffe verwendet werden. Sie dient damit als Vorlage (engl. „default") für die einzelnen Implementierungen.

Viele der in der Datei bereitgestellten Implementierungen für die Microsprachinstruktionen von kaffe konnten nicht direkt in JVM-SEAM verwendet werden. Trotzdem soll durch die Datei gezeigt werden, wie die ursprüngliche Implementierung in kaffe aussah.

#### KaffeAccessors.hh

Die Datei `KaffeAccessors.hh` enthält die Implementierungen, die für JVM-SEAM angepasst werden mussten. Durch `KaffeAccessors.hh` werden die Implementierung in der Datei `DefaultKaffeAccessors.hh` *überschrieben*.

#### kaffe-soft.c

Die Datei `kaffe-soft.c` stellt einige Hilfsfunktionen bereit. Auch wenn die meisten Berechnungen durch die Implementierungen der Microsprache direkt dargestellt werden konnten, gibt es einige komplexere Methoden, die in kaffe schon in externe Implementierungen ausgelagert wurden.

Es handelt sich dabei um komplexere mathematische Operationen wie Typkonversionen und Vergleiche mit Fließkommazahlen. Andere in der ursprünglichen Implementierung in kaffe vorkommende Operationen mit Objekten konnten durch Methoden der JVM-SEAM-Datenstrukturen ersetzt werden.

### 5.2.3 KaffeInterpreter

Der KaffeInterpreter stellt das Bindeglied zwischen dem Ausführungsmodell von SEAM und dem Interpreter-Modell von kaffe dar. Er implementiert die Schnittstelle Interpreter von SEAM, einer spezialisierten Form der Evaluator-Schnittstelle.

In der Abarbeitungsmethode ( KaffeInterpreter :: Run()) übernimmt der KaffeInterpreter zunächst vom übergebenen KaffeFrame die gespeicherten Daten (vgl. Abschnitt 4.5). Diese sind:

- Verweis auf den zu verarbeitenden Bytecode
- Verweis auf die Klasse der zu verarbeitenden Methode
- Die aktuelle Verarbeitungsposition (*program counter*, PC)

Diese werden zum effizienten Zugriff in lokalen Variablen der Abarbeitungsmethode gespeichert. Andere im KaffeFrame gespeicherte Daten werden später direkt aus dem Frame ausgelesen.

Falls vom Scheduler Argumente zur Verarbeitung bereitgestellt werden – entweder als Aufrufargumente oder Rückgabewerte von aus dem Bytecode aufgerufenen Methoden – werden diese in die entsprechenden Speicherbereiche des KaffeFrame übertragen. Die Microsprache von kaffe erwartet Methodenargumente in den lokalen Variablen und Rückgabewerte von aufgerufenen Methoden im Stack des StackFrames.

Mittels bereitgestellter Makros und lokaler Variablen kann die Implementierung der Abbildung auf die Microsprache von kaffe als große Fallunterscheidung bezüglich der aktuellen Bytecode-Instruktion eingebunden werden. Die Makros dienen dazu die Spezifikation der Microsprache in die verschiedenen Fälle (je ein Fall für eine Bytecode-Instruktion) zu expandieren. Die bereitgestellten lokalen Variablen der Abarbeitungsmethode dienen zur Speicherung von Zwischenergebnissen in den einzelnen Fällen. Zusammen mit den lokalen Variablen für PC und Bytecode werden sie hauptsächlich von der Abbildung der Microsprache auf die eigentliche Implementierung in C++ verwendet.

Nach jeder Abarbeitung einer Instruktion wird der PC auf die Position der nächste Instruktion gesetzt. Falls der Scheduler keine Präemptionsbedingung signalisiert, wird die nächste Instruktion verarbeitet. Andere Möglichkeiten zur Unterbrechung oder zum Abbruch der Verarbeitung sind der Aufruf einer neuen Methode, das Erreichen eines Rücksprungpunktes im Bytecode (*RETURN*-Instruktion) oder das Auftreten einer Ausnahme.

Das Aufrufen einer neuen Methode führt dazu, dass dem aktuellen Thread ein neues StackFrame mit der neuen Methode hinzugefügt wird. Das Verlassen der Abarbeitungsmethode führt dann dazu, dass zunächst das neue StackFrame abgearbeitet wird. Danach wird die Verarbeitung mit einem Rückgabewert aus der Verarbeitung des neuen Frames fortgesetzt.

Das Erreichen eines Rücksprungpunktes führt dazu, dass der Rückgabewert über den Scheduler weitergereicht wird und das Frame vom aktuellen Thread gelöscht wird. Die Verarbeitung wird dann im vorher verarbeiteten Frame fortgesetzt.

Eine Ausnahme kann explizit durch die entsprechende Bytecode-Instruktion (ATHROW) oder durch einen Valdierungs- oder Laufzeitfehler im Bytecode (etwa eine leere Referenz) auftreten.

In diesem Fall wird die Abarbeitung abgebrochen. Über die Ausnahme-Verarbeitung des Schedulers von SEAM wird die Ausnahme weitergeleitet. Der Scheduler sucht den nächsten von den Frames des Threads registrierten Ausnahme-Handler und übergibt ihm die weitere Ausführung.

## 5.3 Anpassungen zu Integration des kaffe-Interpreters

Zur leichteren Realisierung der Implementierung der Instruktionen der Microsprache wurden am Datenmodell folgende Anpassungen vorgenommen:

**Methodenbeschreibungen** Die Datenstrukturen zur Methodenbeschreibung wurden um eine Schnittstelle erweitert. Diese erlaubt es, zu prüfen, ob sich die Beschreibung auf eine statische oder nicht-statische Methode bezieht. Im Bytecode-Interpreter des Prototyps wurde diese Prüfung durch die direkte Anforderung einer entsprechenden Datenstruktur geprüft[1]. Die Microsprache von kaffe fordert aber zunächst eine allgemeine Feldbeschreibung an.

**Feldbeschreibungen** Für die Datenstrukturen zur Beschreibung von statischen und nicht-statischen Feldern wurde ebenfalls eine Validierungsschnittstelle analog zur Methodenbeschreibung implementiert.

Außerdem wurden Schnittstellen zur Anforderung des Namens und Typs des Feldes implementiert. Damit ließen sich die entsprechenden Microsprach-Instruktionen zur Bytecode-Verifikation sinnvoll implementieren.

Neben diesen notwendigen Änderungen wurde eine Vielzahl von Debug-Ausgaben zur leichteren Fehlersuche in den Code integriert. Über eine kleine Menge von Makrodefinitionen kann die Menge und Art der Ausgaben sinnvoll gesteuert werden.

## 5.4 Probleme bei der Implementierung

Bei der Integration des kaffe-Interpreters in JVM-SEAM traten zahlreiche Probleme auf. Viele dieser Probleme lassen sich auf die unterschiedlichen Implementierungssprachen C und C++ zurückführen.

Durch das unterschiedliche Datenmodell zwischen kaffe und JVM-SEAM waren an manchen Stellen Kunstgriffe zur Umsetzung nötig. Das Datenmodell von kaffe ist von der Implementierungssprache C geprägt. Die Daten werden in struct- und sogar union-Datenstrukturen gespeichert. Die Anpassung auf das objektorientierte Datenmodell von JVM-SEAM war oft schwierig.

Das größte Problem waren jedoch die überaus vielen Makro-Definitionen. Auch wenn die Makros meist gut dokumentiert und auch in gewisser Weise strukturiert sind, machte die Integration durch gezielte Substitution der Makros viele Schwierigkeiten. Fehler konnten oft nicht

[1] Nur wenn eine Methode statisch ist, wird eine entsprechendes StaticFieldRef -Objekt gefunden.

vom Compiler erkannt oder zumindest nur schlecht angezeigt werden. Oft werden die Makros in kaffe auch zur Umgehung von funktional-imperativen Grenzen benutzt und sorgten damit für Verständnisschwierigkeiten.

Bei der Integration wurden auch einige Fehler in der Abbildung auf die Microsprache von kaffe aufgedeckt. So zeigte sich wiederholt der Fehler, dass nicht korrekt zwischen Wert und Adresse eines Speicherfeldes unterschieden wurde. In der virtuellen Maschine kaffe kommt dieser Fehler nicht zum tragen, da die zugehörigen Makros immer zu einer entsprechenden Variable expandieren, die dann sowohl zur Wertzuweisung als auch zum Auslesen des Wertes geeignet ist. Demgegenüber musste in JVM-SEAM das Makro zum Auslesen durch einen lesenden, das Zuweisung-Makro zu einem schreibenden Zugriff auf das Objekt, das den gewünschten Wert speichert, realisiert werden.

## 5.5 Optimierung

Die anfängliche starke Strukturierung mittels Objekt-Methoden der einzelnen Datenstrukturen wurde nach dem Entfernen der Implementierungsfehler mehr und mehr durch eine schwächere Strukturierung unter Verwendung von statischen Methoden und Makrodefinition ersetzt. Dadurch konnten gerade beim Zugriff auf den Variablen-Stack des KaffeFrame deutliche Leistungsgewinne erzielt werden[2].

Eine weitere Optimierung bestand in der Umstellung der Ganzzahlarithmetik auf die Berechnung mit *tagged integers* [13]. Dabei wird auf eine Demarkierung und Remarkierung der Ganzzahlen aus dem Store verzichtet. Stattdessen wird auf den markierten Ganzzahlen die äquivalente Rechnung durchgeführt.

## 5.6 Umfang der Implementierung

Zur Beurteilung des Umfangs der Implementierung dient die Größe des Programmcodes in Zeilen (engl. „lines of code", LOC). Tabelle 5.1 zeigt die Werte für die verschiedenen Teile. Unterschieden wird dabei in Programmcode aus dem Prototyp, darin geänderte oder neu hinzugefügte Programmteile, Programmcode aus der kaffe-VM und darin modifizierte Teile.

[2] Der verwendete C++-Compiler (GNU Compiler Collection, Version 3.3.4) zeigte hier Schwächen bei der Optimierung, besonders beim *Inlining*. Durch die schwächere Strukturierung konnten diese Mängel teilweise beseitigt werden.

| Programmteil | Lines of Code |
| --- | --- |
| Prototyp | 8500 |
| neue & angepasste Teile | 2500 |
| Programmcode von kaffe | 7000 |
| Anpassungen am kaffe-Code | 1000 |

Tabelle 5.1: Der Umfang der Implementierung in Programmcodezeilen

# 6 Evaluierung

Dieses Kapitel beschreibt die Evaluierung der Implementierung von JVM-SEAM. Es zeigt und erklärt die Ergebnisse.

## 6.1 Benchmarks

Zur Evaluierung der Implementierung wurden verschiedene Standard-Benchmarks verwendet. Diese wurden jeweils auf JVM-SEAM, dem bisherigen Prototyp, der virtuellen Maschine kaffe[1] und der HotSpot JVM[2] ausgeführt. Bei letzteren beiden wurde explizit die Verwendung des Interpreters eingestellt, um vergleichbare Ergebnisse zu erhalten.

Im einzelnen wurden folgende Benchmarks durchgeführt:

***fib*** Berechnung der ersten 31 Fibonacci Zahlen

***tak*** Die Takeushi-Funktion: Aufbau eines Rekursionsbaumes

***nrev*** Reversierung einer Liste mit 3000 Elementen

***quickarray*** QuickSort-Algorithmus auf Feldern

***queens*** Platzierung von 10 Damen auf einem Schachfeld ohne Angriffsmöglichkeit

Die Benchmarks wurden auf einem Dell Inspiron 4100 Notebook mit 768MB RAM und einem 866MHz Pentium-III-Prozessor unter Linux[3] durchgeführt. Jeder wurde insgesamt achtmal durchlaufen, wobei jeweils die Laufzeit in Millisekunden gemessen wurde. Als Ergebnis der jeweiligen Benchmark wurde das arithmetische Mittel der acht Laufzeiten gewertet.

## 6.2 Ergebnisse

Tabelle 6.1 zeigt die Ergebnisse der einzelnen Benchmarks. Für kaffe und HotSpot™ wurden zusätzlich die Benchmarks mit dem JIT-Compiler ausgeführt.

Die Überlegenheit der HotSpot™-JVM ist deutlich erkennbar. Sowohl mit dem Interpreter als auch mit dem JIT-Compiler ist sie in allen Benchmarks ungeschlagen. Der Grund dafür ist die deutlich bessere Qualität dieser kommerziellen JVM für den Produktivbetrieb.

---

[1] kaffe, Version 1.1
[2] HotSpot™, Version 1.4.2_03
[3] Debian/Sarge Linux mit Kernel 2.6.6

| Benchmark | JVM-SEAM | Prototyp | kaffe | HotSpot™ | kaffe-JIT | HotSpot™-JIT |
|---|---|---|---|---|---|---|
| *fib* | 2854 | 2963 | 5424 | 605 | 98 | 44 |
| *tak* | 2616 | 2821 | 5984 | 616 | 90 | 57 |
| *nrev* | 17174 | 13380 | 61628 | 3633 | 12552 | 1415 |
| *quickarray* | 2804 | 3070 | 4300 | 778 | 94 | 85 |
| *queens* | 3355 | 3175 | 7740 | 1069 | 1080 | 360 |

Tabelle 6.1: Ergebnisse der Benchmarks mit den verschiedenen getesteten Java Virtual Machines (Laufzeit in Millisekunden)

Eindeutiger Verlierer ist der Interpreter der kaffe-VM. Am deutlichsten sticht dabei der Benchmark *nrev* heraus. Hier ist kaffe im Interpretermodus über 15-mal langsamer als der HotSpot™-Interpreter und immer noch mehr als fünfmal langsamer als der Prototyp. Selbst mit dem JIT-Compiler ist die kaffe-VM fast 10-mal langsamer als der HotSpot™-JIT und schlägt den Interpreter des Prototyps nur knapp.

JVM-SEAM schlägt sich durchwachsen. Zwar kann die VM in den Benchmarks *fib*, *tak* und *quickarray* den ursprünglichen Prototyp schlagen, muss sich aber in *queens* und *quickarray* dem Prototyp geschlagen geben. Der Benchmark *nrev* zeigt sogar eine Verlangsamung um fast 30%. Dafür wird der kaffe-Interpreter, auf dem die JVM-SEAM basiert, um Faktor 2 übertroffen.

## 6.3 Diskussion

Die Verbesserungen bei *fib*, *tak* und *quickarray* lassen sich auf eine verbesserte Ganzzahlarithmetik gegenüber dem Prototyp zurückführen. Das ständige Umwandeln von Ganzzahlen in ein `word` des SEAM-Store in Verbindung mit der direkten Speicherung in der jeweiligen Zieladresse des Stacks sorgen dafür, dass JVM-SEAM gegenüber dem Prototyp leicht zulegen kann.

*nrev* und *queens* verwenden sehr wenig Ganzzahlarithmetik. In diesen Benchmarks werden hauptsächlich Objekte erstellt und Methoden aufgerufen. Hier zeigt sich die Schwäche des kaffe-Interpreters, der auch in der kaffe-VM in diesen beiden Benchmarks am schlechtesten abschneidet. Besonders bei *nrev* zeigt sich die Schwäche des kaffe-Interpreters im Methodenaufruf. Der Benchmark verwendet Rekursion in solchem Maße, dass eine Anpassung der Stack-Größe der Threads im Quellcode der kaffe-VM nötig war.

Das Ziel der Implementierung einer *effizienten* JVM auf SEAM muss als verfehlt angesehen werden. Auch wenn der kaffe-Interpreter in JVM-SEAM um mehr als Faktor 2 besser abschneidet als in der virtuellen Maschine kaffe selbst, steht fest, dass mit ihm ein Vergleich mit dem Interpreter der HotSpot™-JVM nicht erfolgreich war.

Die Verwendung von SEAM zur Realisierung einer JVM kann jedoch als Erfolg angesehen werden. JVM-SEAM zeigt sich gegenüber der virtuellen Maschine kaffe im Interpreter-Modus als um Faktor 2 effizienter, obwohl beide die gleiche Ausführungseinheit verwenden. Gegenüber

den Prototyp konnte sich JVM-SEAM im arithmetischen Bereich leicht verbessert, muss jedoch im Bereich Methoden-Aufruf Leistung einbüßen. Diese Einbußen sind aber auf den kaffe-Interpreter zurückzuführen. In der virtuellen Maschine kaffe fallen sie wesentlich deutlicher aus.

Am deutlichsten zeigen die Ergebnisse der Benchmarks jedoch die Leistungsfähigkeit von JIT-Compilern gegenüber Interpretern. Zur weiteren Evaluierung von SEAM ist deshalb ein JIT-basierte JVM-Implementierung zu empfehlen.

# 7 Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten zu JVM-SEAM vorgestellt. Dabei handelt es sich hauptsächlich um andere Implementierungen der JVM.

## 7.1 HotSpot$^{TM}$

Die Java Virtual Machine *HotSpot$^{TM}$* von Sun gilt allgemein als die Referenzimplementierung der Spezifikation. Dies bezieht sich nicht nur auf die Korrektheit, sondern auch auf Vollständigkeit und Effizienz.

Da Sun Hotspot$^{TM}$ als kommerzielles Produkt vertreibt, lassen sich nur wenige und eingeschränkte Publikationen zum internen Aufbau finden [19]. Die Ausführungseinheit der JVM basiert auf einer Mischung aus JIT-Compiler und Interpreter. Zunächst wird der Bytecode vom Interpreter ausgeführt. Dabei speichert ein Zähler die Anzahl der Ausführungen. Überschreitet dieser Zähler eine gewisse Grenze, so erkennt die Maschine darin eine schwer-gewichtete Methode, einen *hotspot*. Der Code der Methode wird dann mit dem JIT-Compiler in stark optimierten nativen Code übersetzt und kann somit im Folgenden effizienter als mit dem Interpreter ausgeführt werden.

Da der JIT-Compiler nur für die wichtigsten Codeteile bezüglich der Laufzeit verwendet wird, entsteht keine Leistungseinbuße durch langwierige Übersetzung des Bytecodes. Die *hotspots* des Programmcodes werden trotzdem mit der Effizienz von nativem Code ausgeführt.

Für die vorliegende Arbeit ist HotSpot$^{TM}$ interessant bezüglich der Effizienz. Das Objektmodell eignet sich in den engen Grenzen, in denen es publiziert ist, zum Vergleich und als Studiengrundlage zur Modellierung.

## 7.2 kaffe

Die virtuelle Maschine kaffe dient mit ihrem Interpreter als Grundlage für JVM-SEAM. Sie wurde bereits in Abschnitt 4 und 5 vorgestellt.

## 7.3 SableVM

Die SableVM ist eine Java Virtual Machine auf Open-Source Basis. Hauptziel des Forschungsprojekts ist die effiziente Ausführung von Java-Bytecode [9].

Die Ausführungseinheit der SableVM ist ein Interpreter dessen Hauptroutine über einen Makroprozessor generiert wird. Im Gegensatz zu den meisten anderen Interpretern verwendet er keine Fallunterscheidung pro Instruktion, sondern nutzt die Mnemotics der Bytecode-Instruktionen als direkte Sprungbefehle in der Programmiersprache C. Inzwischen wird neben dem Interpreter auch ein JIT-Compiler für SableVM entwickelt.

Der Forschungsschwerpunkt liegt in der Optimierung des Datenmodells. Dazu wurde ein *bidirektionales Objektlayout* entwickelt. Dazu speichert die VM ausgehend vom referenzierten Ursprung eines Objekts Referenzen und Skalare in zwei Richtungen im Speicher[1]. Zusätzlich werden die Felder jeweils nach der Zugehörigkeit zur Klasse und den Oberklassen geordnet. Dadurch ist es möglich, die Zugriffe auf Objektfelder durch einfache Berechnungen und direkten Speicherzugriff zu realisieren. Der Zugriff auf Methoden eines Objekts wird durch eine ähnliche Anordnung der Datenstrukturen optimiert.

Für die vorliegende Arbeit ist eine Verwendung des bidirektionalen Objektlayouts interessant. Zusätzlich bietet sich ein Vergleich der beiden Interpreter an.

## 7.4 JikesRVM

Die *Jikes Research Virtual Machine* ist eine Java Virtual Machine die als reines Forschungsprojekt gedacht ist [2, 3, 4]. Der ursprüngliche Name des Projekts war *Jalapeño*. Als Forschungsprojekt steht der Code des Projekts unter einer Open-Source Lizenz.

Ziel des Projekts ist die Implementierung einer Java Virtual Machine in der Programmiersprache Java selbst. JikesRVM dient sich damit also selbst als Laufzeitumgebung. Zum einen wurde damit erfolgreich gezeigt, dass dies überhaupt möglich ist. Zum anderen wird daran geforscht, welche neuen Möglichkeiten sich durch diese Architektur für eine Java Virtual Machine ergeben.

Durch die Architektur bedingt ist ein Interpreter aus Ausführungseinheit der JikesRVM nicht möglich[2]. Dafür gibt es zwei verschiedene JIT-Compiler. Der *Baseline Compiler* ist für eine schnelle Übersetzung des Bytecodes in nativen Code zuständig. Er führt fast keine Optimierung durch. Wie in HotSpot$^{TM}$ versucht JikesRVM die Laufzeit-kritischen Methoden zu finden. Diese werden dann mit dem *Optimizing Compiler* zu stark optimiertem nativen Code übersetzt.

Die Implementierung der VM selbst in Java führt dazu, dass der *Optimizing Compiler* auch den Code der Maschine selbst adaptiv optimieren und sogar zum *Inlining* in den auszuführenden Programmcode verwenden kann. Ein effizientes Datenmodell sorgt ebenfalls für eine gute Leistung.

[1] Eine Objektreferenz zeigt also nicht wie allgemein üblich auf den Anfang des allokierten Speicherbereichs, sondern der Speicherbereich wird ab dem angegebenen Adresse in beide Richtungen bis zu seinen Grenzen genutzt.

[2] Ein Interpreter bildet die Instruktionen auf Primitive der Implementierungssprache ab. Bei JikesRVM entstünde damit aber ein Henne-Ei-Problem, da die Implementierung sich selbst ausführen muss.

Da sich die Architektur der JikesRVM implementierungsbedingt stark von der Architektur von JVM-SEAM unterscheidet, kann die JikesRVM nur als Referenzimplementierung zur Evaluierung verwendet werden.

## 7.5 Microsoft CLR

Microsoft entwickelt mit der *Common Language Runtime Environment* – kurz *CLR* – eine virtuelle Maschine, die viele Gemeinsamkeiten mit einer JVM hat [18]. Sie wird als Teil der *.NET*-Projekts für eine Vielzahl von Programmiersprachen entwickelt.

Der Programmcode der verschiedenen Programmiersprachen wird dazu in den Bytecode der CLR übersetzt. Der Code ist ähnlich dem Java-Bytecode Objekt-basiert, was der Mehrzahl der auf ihn abzielenden Programmiersprachen zugute kommt, die ebenfalls Objekt-orientiert sind. Demzufolge ist auch das Datenmodell der CLR auf die Handhabung von Objekten als hauptsächlichen Datenstrukturen ausgelegt.

## 7.6 Weitere verwandte Arbeiten

Das Datenmodell, insbesondere das Objektlayout, einer Java Virtual Machine ist für eine effiziente Implementierung überaus wichtig. In der Literatur finden sich häufig Abhandlungen, die sich ausschließlich mit diesem Thema allgemein ohne konkreten Bezug zu einer spezifischen JVM beschäftigen [5, 6]. Daneben findet sich auch Literatur zu Themen wie dem Klassenlader [14].

# 8 Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurden Entwurf und Implementierung von JVM-SEAM vorgestellt. Außerdem wurde die Implementierung in Bezug auf Leistungsfähigkeit mit anderen JVMs und in Bezug auf Verwendbarkeit von SEAM zur Realisierung einer Java Virtual Machine evaluiert.

## 8.1 Zusammenfassung

Die Verwendung von SEAM für verschiedene Dienste der JVM ist im Bezug auf die Effizienz als Erfolg zu bezeichnen, wie die Diskussion der Benchmarks belegt. Die ähnliche Leistung von JVM-SEAM gegenüber dem Prototyp zeigt, dass SEAM zur Realisierung von JVMs geeignet ist. Dies belegt auch die deutlich bessere Leistung des kaffe-Interpreters in JVM-SEAM gegenüber der virtuellen Maschine kaffe.

Die Modellierung von JVM-SEAM wurde durch die Verwendung von SEAM deutlich vereinfacht. Nicht vorhandene Dienste, wie der notwendige Klassenlader, ließen sich mit dem SEAM-Framework auch einfach realisieren und konnten nahtlos in das Modell integriert werden.

Bei der Implementierung von JVM-SEAM war die Verwendung der Hochsprache C++, in der SEAM implementiert ist, von Vorteil. Die Modelle konnten so leicht und strukturiert umgesetzt werden. Die Optimierung der Implementierung hat jedoch in einigen Fällen gezeigt, dass eine zu starke Strukturierung der Leistungsfähigkeit gegenläufig ist.

JVM-SEAM hat mit dem Interpreter der virtuellen Maschine kaffe nicht an die Leistungsfähigkeit der HotSpot$^{TM}$-JVM heranreichen können. Jedoch zeigte sich im arithmetischen Bereich eine leichte Verbesserung gegenüber dem Prototyp. Die Leistungseinbußen im Bereich Methodenaufrufe können anhand der Ergebnisse der Benchmarks auf den integrierten Interpreter der virtuellen Maschine kaffe zurückgeführt werden.

## 8.2 Ausblick

Wie die Ergebnisse der Benchmarks gezeigt haben, ist allein durch Verwendung von nativem Code eine deutliche Leistungssteigerung zu erwarten. Dazu bedarf es der Integration eines JIT-Compilers als Evaluator.

Als erstes wäre hier auf den JIT-Compiler der virtuellen Maschine kaffe zu verweisen [12]. Er bietet zum Einen den Vorteil, dass er recht leistungsfähig ist, wie die entsprechenden Messwerte der Benchmarks belegen.

Zum Anderen ist eine Integration des kaffe-JIT-Compilers höchstwahrscheinlich deutlich einfacher als die eines anderen, da JIT-Compiler und Interpreter in kaffe auf dem gemeinsamen Modell der Microsprache aufbauen, die bereits in JVM-SEAM integriert ist.

Andere JIT-Compiler könnten sich allerdings als noch leistungsfähiger erweisen als kaffe, das Leistungsmängel im Bereich Methodenaufruf gezeigt hat. Hier bietet sich der relativ neue JIT-Compiler der SableVM [9] an. Auch eine Portierung des adaptiven *Optimizing Compilers* der JikesRVM [4] nach C++ wäre eventuell möglich.

Neben der weiteren Optimierung der Ausführung könnten auch andere Komponenten durch Verbesserungen zu einer nicht zu unterschätzenden Leistungssteigerung beitragen. Das Datenmodell von JVM-SEAM ist wenig optimiert sondern spiegelt eher eine geradlinige Implementierung der Spezifikation wieder. In der Literatur finden sich verschiedene Studien zum Datenmodell einer JVM, die sich mit schnellerer Objektsperrung [6, 17] oder der effizienten Speicheranordnung der Objekte [5, 9] beschäftigen. Auch diese Möglichkeiten sollten untersucht werden.

# Abbildungsverzeichnis


2.1 Die Komponenten einer Java Virtual Machine und ihr Zusammenwirken . . . . 4
2.2 Die Bytecode-Instruktionen zum Addieren zweier Ganzzahlen . . . . . . . . . 7
2.3 Die Schnittstelle der Wurzel-Klasse Object der Programmiersprache Java . . . . 8
2.4 Der allgemeine Aufbau eines Objekts im Speicher . . . . . . . . . . . . . . . . 9
2.5 Die Funktionsweise eines Schedulers . . . . . . . . . . . . . . . . . . . . . . . . 13

3.1 Die Architektur von SEAM . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2 Das idealisierte Ausführungsmodell von SEAM . . . . . . . . . . . . . . . . . . 18

4.1 Die Komponenten der JVM-SEAM . . . . . . . . . . . . . . . . . . . . . . . . 22
4.2 Der Aufbau eines Stack-Frames . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.3 Die Bytecode-Abstraktion der virtuellen Maschine kaffe . . . . . . . . . . . . 28

5.1 Die Komponenten zur Implementierung des Interpreters . . . . . . . . . . . . . 31
5.2 Ein Ausschnitt der Bytecode-Abbildung auf die Microsprache von kaffe . . . . 32

# Tabellenverzeichnis


2.1 Die skalaren Datentypen einer JVM . . . . . . . . . . . . . . . . . . . . . . . . 6

4.1 Abbildung der skalaren Datentypen auf Datentypen des Store . . . . . . . . . . . 23

5.1 Der Umfang der Implementierung in Programmcodezeilen . . . . . . . . . . . . 37

6.1 Ergebnisse der Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

# Literaturverzeichnis


[1] IEEE/ANSI Std. 754. IEEE Standard for Binary Floating-Point Arithmetic, 1985.

[2] B. Alpern. C. R. Attanasio. J. J. Barton. M. G. Burke. P. Cheng. J.-D. Choi. A. Cocchi. S. J. Fink. D. Grove. M. Hind. S. F. Hummel. D. Lieber. V. Litvinov. M. F. Mergen. T. Ngo. J. R. Russell. V. Sarkar. M. J. Serrano. J. C. Shepherd. S. E. Smith. V. C. Sreedhar. H. Srinivasan und J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1): Seiten 211–238, 2000.

[3] Bowen Alpern. C. R. Attanasio. Anthony Cocchi. Derek Lieber. Stephen Smith. Ton Ngo. John J. Barton. Susan Flynn Hummel. Janice C. Sheperd und Mark Mergen. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 1999)*, Seiten 314–324, Denver, Colorado, USA, 1999. ACM Press.

[4] Matthew Arnold. Stephen Fink. David Grove. Michael Hind und Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, Seiten 47–65. ACM Press, 2000.

[5] David F. Bacon. Stephen J. Fink und David Grove. Space- and Time-Efficient Implementation of the Java Object Model. In Boris Magnusson, Editor, *Proceedings of the Sixteenth European Conference on Object-Oriented Programming (ECOOP 2002)*, Volume 2374 von *Lecture Notes in Computer Science*, Seiten 111–132, Málaga, Spanien, Juni, 2002. Springer-Verlag.

[6] David F. Bacon. Ravi Konuru. Chet Murthy und Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the ACM SIGPLAN '98, Conference on Programming Language Design and Implementation (PLDI)*, Seiten 258–268. ACM Press, Juni, 1998.

[7] Thorsten Brunklaus und Leif Kornstaedt. A Virtual Machine for Multi-Language Execution. Technical Report, Programming Systems Lab, November, 2002. http://www.ps.uni-sb.de/Papers/abstracts/multivm.pdf (August 2004).

[8] Patrick Doyle und Tarek Abdelrahman. Jupiter: A Modular and Extensible JVM. In *Proceedings of the Third Annual Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing*, Seiten 37–48. ACM, Juni, 2001.

[9] Etienne M. Gagnon und Laurie J. Hendren. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *Java(TM) Virtual Machine Research and Technology Symposium*, Seiten 27–40, 2001.

[10] GNU Classpath - A set of essential libraries for supporting the Java programming language, 2004. http://www.gnu.org/software/classpath/ (August 2004).

[11] Richard Jones und Rafael Lins. *Garbage Collection*. Wiley, 1996.

[12] kaffe - A free Java Virtual Machine, 2004. http://www.kaffe.org/ (August 2004).

[13] Xavier Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report RT-0117, Institute National de Recherche en Informatique et en Automatique, Februar, 1990.

[14] Sheng Liang und Gilad Bracha. Dynamic Class Loading in the Java™ Virtual Machine. In *Proceedings of the 1998 ACM SIGPLAN, Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98)*, Seiten 36–44. ACM Press, 1998.

[15] Tim Lindholm und Frank Yellin. *The Java™ Virtual Machine Specification*. Adison Wesley, 2. Edition, 1999.

[16] Sun Microsystems. *Java 2 SDK Documentation: Object Serialization*. Sun Microsystems, 1.4.1. Edition, 2002.

[17] Tamiya Onodera und Kiyokuni Kawachiya. A study of locking objects with bimodal fields. In *Proceedings of the 1999 ACM SIGPLAN, Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99)*, Volume 34.10 von *ACM Sigplan notices*, Seiten 223–237. ACM Press, November, 1999.

[18] David Stutz. Ted Neward und Geoff Shilling. *Shared Source CLI Essentials*. O'Reilly, 1. Edition, März, 2003.

[19] Sun Microsystems. The Java HotSpot(TM) Virtual Machine. Technical White Paper, Sun Microsystems, September, 2002.

[20] Bill Venners. *Inside the Java 2 Virtual Machine*. McGrayHill, 2. Edition, 1999.

# Index

.NET, 45
**J**ust-**I**n-**T**ime (JIT) Compilation, 12

Abstrakte Plattform, 3, 27
Ausführungseinheit, 22
automatische Speicherbereinigung, 3, 10, 17, 20, 19

Baseline Compiler, 44
bidirektionales Objektlayout, 44
Bindung, 5
blocks, 17
Boxing, 23
BuildClassFrame, 26
BuildClassWorkers, 26

chunks, 17
Class-Dateien, 4
Closure, 17, 26
Common Language Runtime Environment, 45
Copying Garbage Collector, 10
Corba, 20

Future, 26

Garbage Collector, 10
green threads, 13

HotSpot™, 43

Inlining, 36, 44
Interface, 4

Jalapeño, 44
Java Security, 5, 10
Jikes Research Virtual Machine, 44

KaffeInterpreter, 25
Keller, 7

language layer, 15
lazy binding, 26

Microsprache, 27
Mnemonic, 7

native threads, 13, 24

Optimizing Compiler, 44, 48

Pickling, 19, 20
Plugins, 15
pop, 12, 18
program counter, 25, 27, 34
push, 12, 18

Reflection, 11

Serialisierung, 20
späte Bindung, 26
Stack, 7, 25
Stapel, 7, 25
Store, 15
System Properties, 10

tagged integers, 36

Unpickling, 20

virtuelle Maschine, 1
VM, 1

worker, 19