Laufzeitverhalten und Portierungsaspekte der Java Virtual Machine (JVM) und ausgewählter Java-Bibliotheken

Marc Schanne

Eine Studienarbeit am
Institut für Programmstrukturen und Datenorganisation
Betreuer: Dr. Michael Philippsen (IPD), Ralf J. Schäfer (Siemens AG, München)

Universität Karlsruhe

26. Februar 1997

Zusammenfassung

Die Programmiersprache Java von Sun Microsystems hat innerhalb kürzester Zeit die schön geordnete Welt der Softwareentwicklung durcheinander geworfen. Objektorientiert, sicher, robust und portabel, soll das Java-System die Entwicklungsplattform der Zukunft sein.

Eine Komponente des Java-Systems ist die Java Virtual Machine (JVM), die Spezifikation für eine abstrakte stack-basierte Maschinenstruktur.

Diese Arbeit stellt das Modell der JVM vor und untersucht und vergleicht die Möglichkeiten und Grenzen von JVM auf verschieden Plattformen. Das Laufzeitverhalten unterschiedlicher JVM-Implementierungen wird anhand einer Sammlung von Testprogrammen gemessen und verglichen. Zusätzlich sind Konzepte wie Just-in-Time-Übersetzung oder die Verwendung von Native-Routinen zur beschleunigten Ausführung von Programmen Inhalt der Überlegungen.

In einem zweiten Teil wird das Abstract Window Toolkit (der JDK 1.0.2) als größte Java Bibliothek betrachtet und Möglichkeiten der Portierung auf andere Hardwareplattform und Betriebssystem untersucht.

Quem in ipsa re trepidare nolueris, ante rem exercas - SENECA

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Java-Benchmarks

Der Siegeszug der Programmiersprache Java steht in engem Zusammenhang mit der Möglichkeit mit ihr plattformunabhängige Programme für das Internet zu entwickeln. Diese sind als *Applets* im Fenster eines World Wide Web-Browsers ausführbar. Mit Applets lassen sich nicht nur Web-Seiten interaktiver gestalten, sondern es ist möglich beliebige Programme auszuführen. In Folge einer richtigen Java-Euphorie sind deshalb auch in Java geschriebene Benchmark-Applets entstanden. Es genügt der Abruf einer URL (Uniform Resource Locator) und schon kann man die Leistungsfähigkeit seines eigenen Systems testen.

1.1.1 Beispiele im WWW

Im Folgenden sind die drei derzeit wichtigsten Java-Benchmarks aufgelistet:

- Linpack[1], von JACK DONGARRA und REED WADE, ist schon lange als Fortran oder C/C++-Benchmark zu haben [Dong96], und die Java-Version ist deshalb besonders ausgereift. Auch wenn der direkte Vergleich mit den Leistungszahlen herkömmlicher Compiler schwierig ist, ermöglichen die numerischen Tests den Performance-Vergleich verschiedener JVM.

- CaffeineMark[2] und der Java Performance Report ist der derzeit wohl am häufigsten zitierte Test, wenn es um Java und die JVM geht. Auf die Ergebnisse von CaffeineMark 2.5 geht der Abschnitt 3.6 ein.

- BenchBeans[3] — von HOLGER RECKTER — ist ein weiterer Benchmark, der versucht, alle Aspekte der Java-Performance, einschließlich Grafik zu behandeln. 13 Einzeltests ermitteln Vergleichswerte für Arithmetik, Grafik, Speicher-Allozierung, Methoden und Text.

[1] http://www.netlib.org/benchmark/linpackjava/
[2] http://www.webfayre.com/pendragon/cm2/caffainemark2.html
[3] http://user.cs.tu-berlin.de/˜mondraig/english/benchbeans.html

Eine gute Übersicht aller WWW-Ressourcen zum Thema gibt es bei JONATHAN HARDWICK[4].

1.1.2 Möglichkeiten und Grenzen

Durch das Sammeln möglichst vieler Ergebnisse versuchen sämtliche Web-Java-Benchmarks eine Vergleichsmöglichkeit verschiedener Rechnerplattformen und Interpreter zu erreichen. Die Testbedingungen sind bei dieser Vorgehensweise natürlich nicht immer klar festgelegt, und deshalb sind die Ergebnisse auch nur eingeschränkt brauchbar. Zumindest läßt sich so aber die Leistung einer JVM einordnen.

1.2 Just in Time-Compiler (JITC)

Um die Maschinenunabhängigkeit eines Java-Programms zu gewährleisten, erzeugt der Compiler ein architekturneutrales Objektdateiformat. Dieser *Bytecode* wird dann vom Java-Laufzeitsystem ausgeführt (vgl. Abschnitt 2). Er wird interpretiert, und dies bedingt, daß die Ausführgeschwindigkeit langsamer ist, als die eines in C geschriebenen, übersetzten und gebundenen Programms.

Ein Versuch diesen Nachteil auszugleichen sind sogenannte *Just in Time*-Übersetzer (JITC), die den zu interpretierenden Java-Bytecode direkt vor der Ausführung in nativen Code übersetzen und so auch Hardware- oder systembedingte Vorteile nutzen können (vgl. dazu Abschnitt 4.2). Diese Studienarbeit untersucht in Kapitel 3, die so zu erreichende Leistungssteigerung gegenüber der einfachen Interpretation zu ermitteln.

1.3 Laufzeitintensive Anwendungen

Trotz ständiger Veränderungen beruht die Existenzberechtigung vieler Computersysteme darauf, daß sie in der Lage sind, schnelle und genaue numerische Berechnungen zu realisieren. Auch die in dieser Studienarbeit ausgewerteten Benchmarks (vgl. Abschnitt 3.2) gehen deshalb von mathematischen Objekten aus: Matrizen, Polynome und Listen und bewerten die Rechenzeit für verschiedene Routinen, wie Matrixmultiplikation oder Sortieren von Listen.

[4] http://www.cs.cmu.edu/~jch/java/optimization.html

Kapitel 2

Das Java-System

Java ist mehr als nur eine Programmiersprache, sie ist vielmehr ein „Technologiekonzept“ [Dahl97]. Das Java-System beinhaltet ebenfalls ein Konzept für ein abstraktes Maschinenmodell und eine Spezifikation der Kernbibliotheken. Die folgenden Abschnitte stellen diese Konzepte kurz vor und zeigen anhand des Java Development Kits 1.0.x (JDK) von JavaSoft wie diese Modelle real implementiert werden können.

2.1 Die Programmiersprache

Die Sprache Java wurde seit 1990 bei Sun Microsystems entwickelt. In ihrer Syntax ähnelt sie sehr C und C++ und ist vollständig objektorientiert. Für weitergehende Informationen gibt es von Sun *The Java Language Specification* [Sun95a] und *The Java Language: A White Paper* [Sun94]. Für eine Einführung mit Referenz ist [Flan96] gut geeignet.

2.2 Die Java Virtual Maschine

Zusätzlich zur Spezifikation der Sprache hat Sun Microsystems auch *The Java Virtual Maschine Specification* [Sun95b] festgelegt.

Die JVM ist eine abstrakte stack-basierte Maschine und besteht aus folgenden Komponenten:

- Maschienenbefehlssatz
- Register
- Laufzeitumgebung
- Datenspeicher (Stack, Heap)

Zur Implementierung sagt Sun folgendes: *All of these are logical, abstract components of the virtual machine. They do not presuppose any particular implementation technology or organization, but their functionality must be supplied in some fashion in every Java system based on this virtual machine* [Sun95b].

In der Spezifikation wird deshalb auch nur die Funktionalität und Begrenzung der Systemkomponenten beschrieben. Eine erweiterte Darstellung wird in [TiYe96] und [Dahl97] gegeben.

Die einzige Festlegung, die für einen konkreten Interpreter von Java-Bytecode getroffen wird, ist folgender Pseudo-Code-Abschnitt:

```
do {
  fetch a byte
  execute an action depending on
    the value of the byte
} while (there is more to do);
```

Der Interpreter erhält eine Folge von übersetztem Java-Code und führt byteweise die Befehle der virtuellen Maschine aus.

Teil der JVM-Spezifikation ist auch die Festlegung des *Class File Format*, in dem der Bytecode abgelegt ist. Es ermöglicht den Austausch von übersetztem Code mit jeder beliebigen Plattform.

2.3 Die Kernbibliotheken

Mit der Entwicklung von *Kernbibliotheken*[1] und der Schnittstellen-Spezifikation versucht JavaSoft eine auf allen Plattformen gleiche Java-Umgebung (vgl. auch [Sun95c]) festzulegen.

Das in Abschnitt 5.1 näher beschriebene *Abtract Window Toolkit* (AWT) ist eine der ersten Bibliotheken und schon in der JDK 1.0.x enthalten. Ab der Version 1.1 sollen noch andere in die Standard-Distribution aufgenommen werden.

2.4 Die Implementierung

Abbildung 2.1 skizziert die Zusammenhänge der einzelnen Komponeten des Java-Systems.

Der Java-Compiler übersetzt Java-Code in Bytecode, entsprechend der JVM-Spezifikation. Um diesen auszuführen, werden dann ein Java-Interpreter und die Kernbibliotheken gebraucht. Um auf einer beliebigen Plattform eine vollständige Java-Laufzeit-Umgebung zu unterstützen, müssen die gestrichelt gezeichneten Teile implementiert werden.

Am Beispiel des Java Development Kits (JDK) von JavaSoft wird im nächsten Abschnitt eine Implementierung der JVM vorgestellt. Die JVM-Spezifikation läßt sich aber auch als Hardware realisieren; ein Beispiel dafür ist der *Pico-Chip* von Sun Microsystems.[2]

Im Folgenden wird der Begriff Java Virtual Maschine (JVM), wie im allgemeinen Sprachgebrauch üblich, für die Implementation der Spezifikation verwendet.

[1] http://www.javasoft.com/products/apiOverview.html

[2] http://www.sun.com/sparc/java/

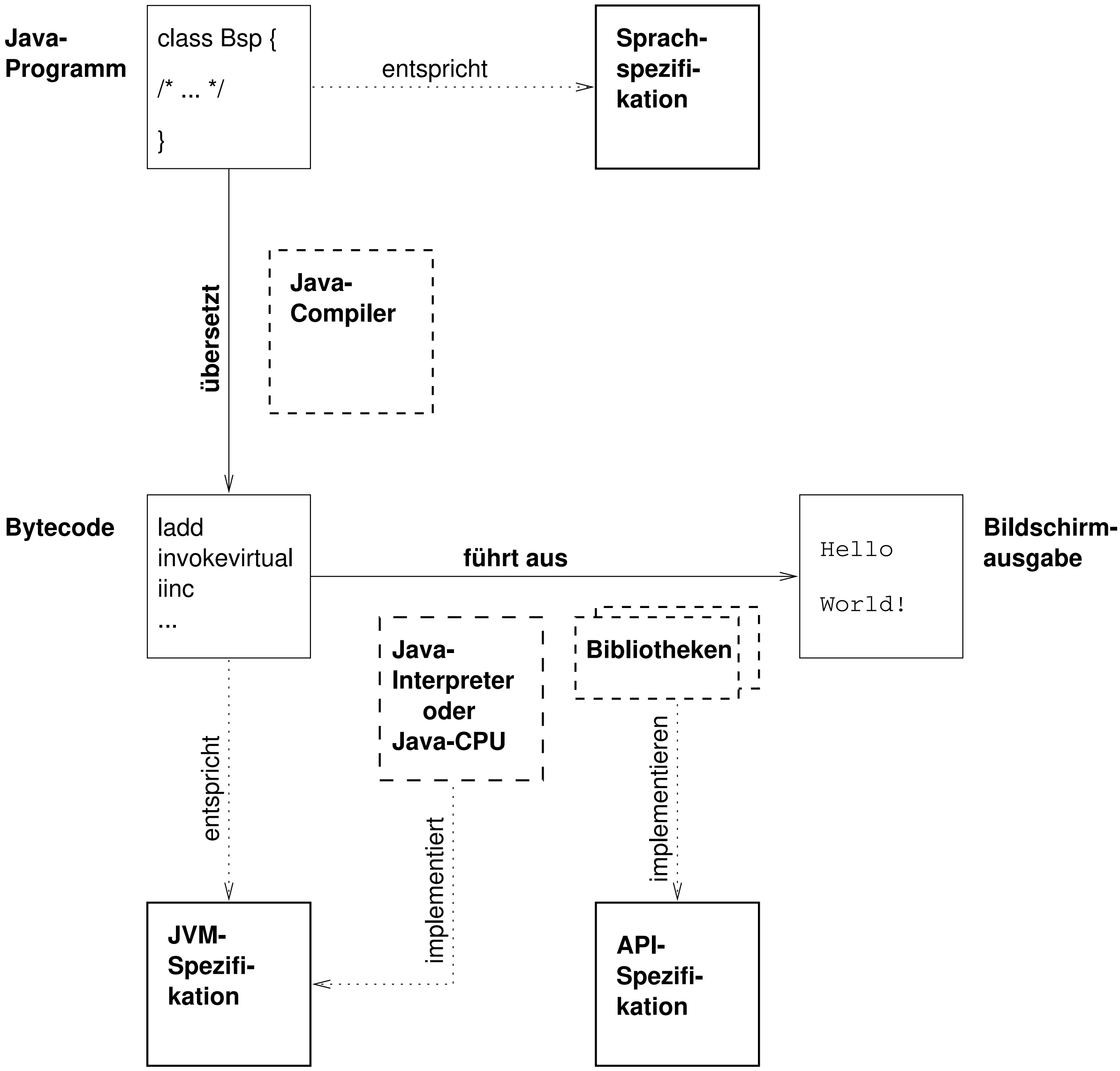

Abbildung 2.1: Komponeten des Java-Systems

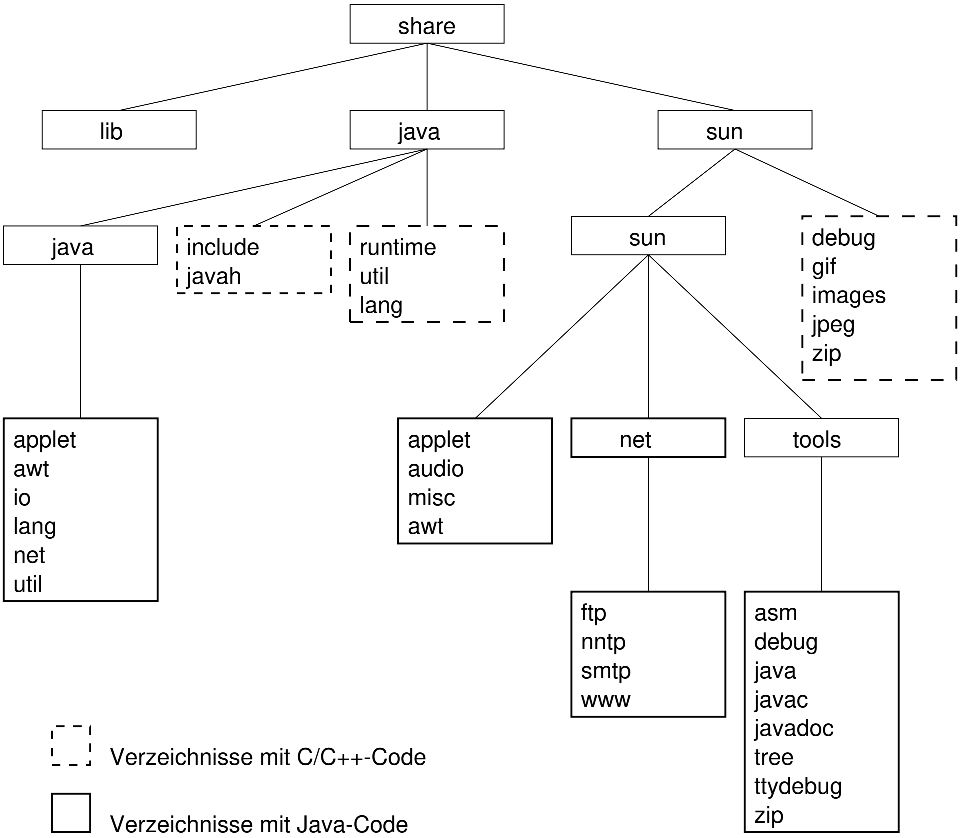

Abbildung 2.2: Verzeichnisstruktur JDK 1.0.2 für Windows: `share`-Zweig

2.5 JDK 1.0.x

Die JDK 1.0.x von JavaSoft wurde gemäß der JVM-Spezifikation entwickelt. Sie ist für Windows 32 Bit Plattformen und für Solaris erhältlich.[3]

Der Quellcode ist in einen plattformunabhängigen (vgl. Abbildung 2.2) und in einen -abhängigen Teil (vgl Abbildung 2.3) gegliedert. Beide besitzen ein `java`- und ein `sun`-Unterverzeichnis.

In den beiden `java`-Verzeichnissen sind alle Funktionen der Sprache und des Interpreters gesammelt. In `java\runtime`, `java\util` und `java\jang` sind die Hauptteile der JVM implementiert. Die Verzeichnisse `java\java` enthalten nur Java-Code; aufbauend auf Funktionen in C (z. B. in `java\runtime`) werden hier die Klassen der Kernbibliotheken implementiert oder Schnittstellen definiert.

Der `sun`-Zweig in beiden Abbildungen ist ähnlich aufgebaut. In den direkten Unterver-

[3]Die Executeables sind bei JavaSoft (http://www.javasoft.com) frei erhältlich. Aber auch der Source-Code ist unter bestimmten Bedingungen (http://java.sun.com/nav/business/license_dl.html) frei erhältlich. JavaSoft fördert so die Portierung auf andere Systemplattformen. Nach einer Funktions-Kontrolle kann von jeder Portierung die Übersetzung frei zur Verfügung gestellt werden.

Für Firmen gibt es die Möglichkeit der kommerziellen Lizensierung.

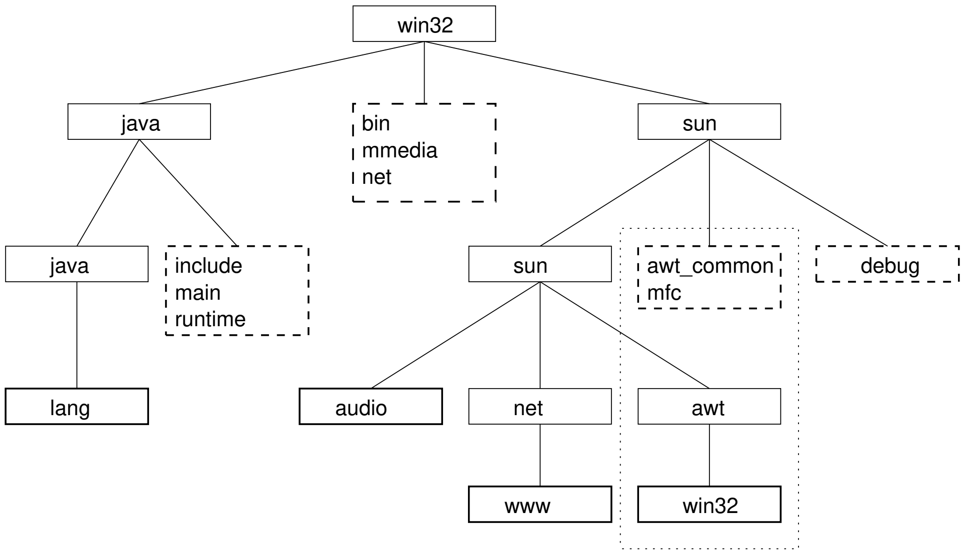

Abbildung 2.3: Verzeichnisstruktur JDK 1.0.2 für Windows: `win32`-Zweig

zeichnissen von `sun` (`debug`, `gif`, `images`, ... und `awt_common`, `mfc`, ...) wird in C/C++ Funktionalität implementiert, auf die in den Java-Klassen in `sun\sun\*` zugegriffen wird. Sun/Javasoft bietet hier neben der Implementierung für das AWT auch noch zusätzliche Tools, wie einen Debugger und Compiler. Der Java-Compiler ist direkt in Java geschrieben (`sun.tools.javac.Main`) und kann deshalb mit jeder JVM-Implementierung verwendet werden.

Die in Abbildung 2.3 zusätzlich markierten Verzeichnisse beinhalten den plattformabhängigen Programmcode für das AWT, dessen Struktur und Portierung in Abschnitt 5.1 näher betrachtet wird.

Die hier beschriebene Struktur der JDK 1.0.x stellt nur eine Implementierung der JVM-Spezifikation dar, aber viele Software-Hersteller haben ihre Produkte auf diese aufgebaut.

Kapitel 3

Laufzeitverhalten verschiedener JVM

Die Java Virtual Machine (JVM) spielt eine zentrale Rolle bei der Portabilität von Java. Sie stellt eine Abstraktionsschicht zwischen übersetztem Java-Programm (Bytecode) und der zugrundeliegenden Hardwareplattform mit Betriebssystem zur Verfügung. Die Abbildung 3.1 stellt diese Modell vereinfacht dar.

3.1 Allgemeine Überlegungen

Ziel der Testprogramme ist es, die Leistungsfähigkeit von JVM-Implementierungen auf verschiedener Hardware und Betriebssystemen zu vergleichen.

Die Benchmark Sammlung (Anhang A) beinhaltet mathematische Problemstellungen, wie das Primzahlensieb des ERATHOSTENES oder die Multiplikation von Matrizen und Polynomen. Die Performance des Benutzerinterfaces oder auch die grafische Leistungsfähigkeit werden nicht betrachtet. Hier hängen bessere Testergebnisse meist mehr vom zugrundeliegenden System als von der JVM-Implementierung ab. X-Window und andere Client/Server-Grafik-Subsystemen geben anders, wie zum Beispiel die Grafikoberfläche von Windows 95, schon vor der kompletten Anzeige grafischer Elemente die Kontrolle an das Programm zurück. Dies verhindert eine genaue Zeitmessung und auch der Vergleich dieser beiden grundlegend verschiedenen Systeme ist schwer möglich.

Bei den einzelnen Benchmarks wurde Wert darauf gelegt, unterschiedliche Programmiertechniken (Rekursion, große Arrays, etc.) zu verwenden.

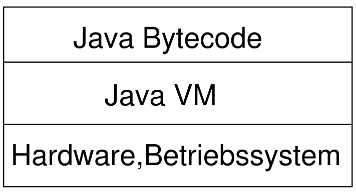

Abbildung 3.1: Abstrakte Ebenen bei Java

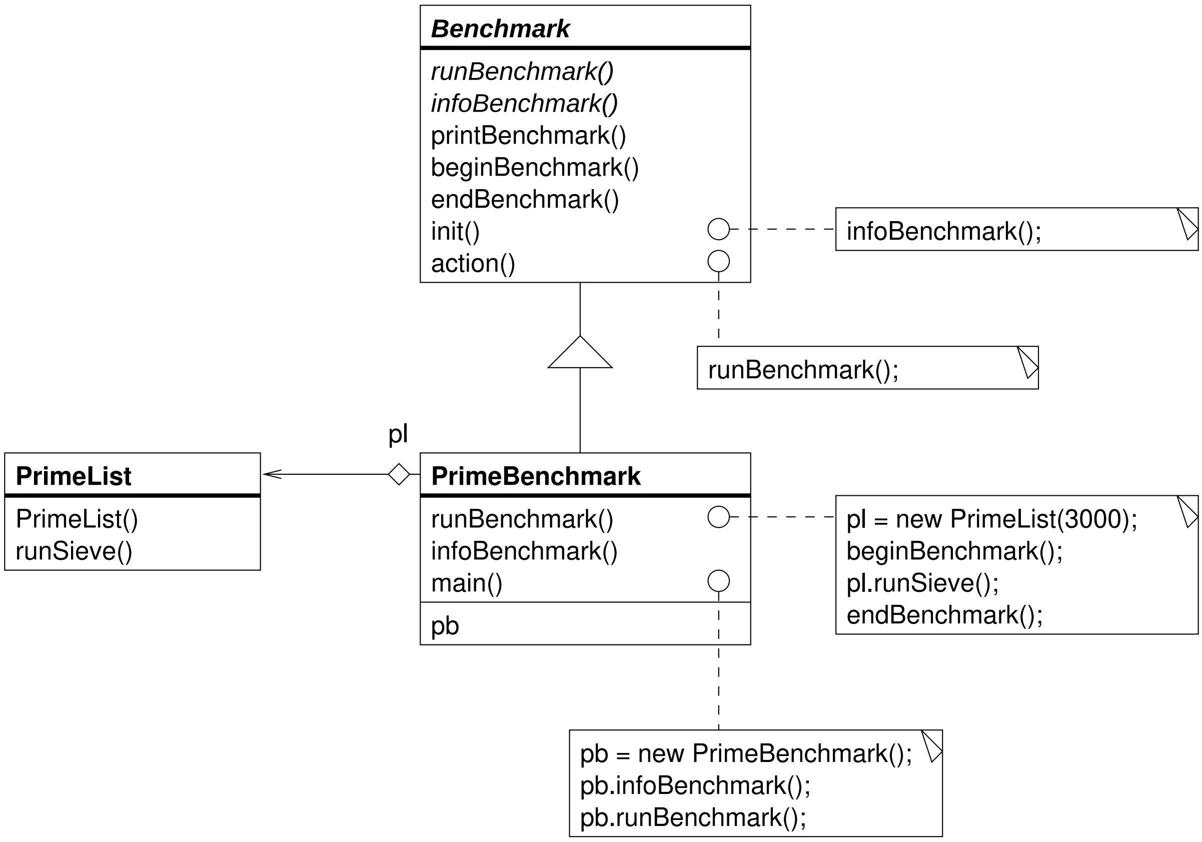

Abbildung 3.2: OMT-Diagramm: `PrimeBenchmark`

3.2 Benchmark-Programme

Das Diagramm[1] in Abbildung 3.2 zeigt beispielhaft am Primzahlen-Benchmark (`PrimeBenchmark`) die objektorientierte Struktur der Benchmark-Klassen.

`Benchmark` ist eine abstrakte Klasse, von der alle konkreten Testprogramme erben. Sie stellt Methoden zur systemunabhängigen Ausgabe und Zeitmessung zur Verfügung. `printBenchmark()` leitet die Ausgabedaten bei einem Applet in ein Textfeld (`java.awt.TextArea`), bei einer Applikation an die Standardausgabe um. Die beiden Methoden `beginBenchmark()` und `endBenchmark()` sind eine Zeitklammer um den Benchmark, stoppen Start- und Endzeit, und geben die Laufzeit aus.

Die abstrakt definierten Routinen `runBenchmark()` und `infoBenchmark()` müssen von jedem Benchmark selbst implementiert werden. `infoBenchmark()` gibt den Name und die Versionsnummer des Tests aus, und in `runBenchmark()` wird der Test ausgeführt.

[1] Die Klassendiagramme in dieser Studienarbeit basieren auf OMT, der Object Modeling Technique, wie sie in Auszügen in [GAMM96] beschrieben wird.

`MatrixBenchmark`

implementiert eine einfache *Matrizenmultiplikation*:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

und den GAUSS-Algorithmus zum Lösen von *Linearen Gleichungssystemen* (LGS):

$$A^{(n,n)} x = b$$

Der Algorithmus für die Multiplikation berechnet durch einfaches Aufsummieren in drei verschachtelten Schleifen die Lösungsmatrix. Bei einer quadratischen (n,n)-Matrix bedeutet dies einen Aufwand von $O(n^3)$. Für den Benchmark wird ein Produkt zweier (200,200)-Matrizen, mit zufällig gesetzten `double`-Werten, berechnet.

Das LGS wird mit dem Gaußschen Eliminationsverfahren, wie es beispielsweise in [Sedg91.1] beschrieben wird, gelöst. Der Test führt dies für ein Gleichungssystem mit 400 Gleichungen und 400 Unbekannten — in `double` — durch.

`PolyBenchmark`

multipiziert zwei Polynome (Instanzen der Klasse `Poly`) nach dem Algorithmus von KARAZUBA. Dieser Algorithmus (vgl. [Sedg91.2]) löst nach dem Prinzip *Teile und Herrsche* rekursiv das Problem für Polynome (Objekte der Klasse `Poly`) der Länge n mit einem Aufwand in $O(n^{1,58})$. In `PolyBenchmark` ist $n = 4000$ gewählt.

`PrimeBenchmark`

ist der im obigen Diagramm dargestellte Test. Als Datenstruktur verwendet er eine `boolean`-Flag-Liste für alle Zahlen. Der verwendete Algorithmus nach ERATOSTHENES filtert Nichtprimzahlen aus, indem er für alle Primzahlen von 2 beginnend die Vielfachen aus der Liste streicht (auf `false` setzt). Wenn er alle Zahlen bis $\sqrt{n}$ (mit n als Länge der Liste, hier 9.000.000) getestet hat, sind nur noch Primzahlen `true`.

Der Primzahlentest ist zweigeteilt, einmal wird die Zeit zur Erzeugung des `PrimeList`-Objekts, das im wesentlichen ein `boolean`-Array der Länge 9.000.000 ist, gemessen und zum zweiten die Ausführungszeit des eigentlichen *Primzahlensieb*-Algorithmus.

`SortBenchmark`

erzeugt ein `SortList`-Objekt, eine Liste von Zahlen des Typs Integer mit Sortieralgorithmen. Die Liste wird mit Zufallszahlen gefüllt und anschließend mit *Shell-Sort* sortiert. *Shell-Sort* (vgl. [Sedg91.3]) ist eine Verbesserung des *Sortierens durch Einfügen*, indem es Teillisten vorsortiert.

Inhalt des Tests ist einmal der häufige Zugriff auf die 1000 Array-Elemente während des Sortiervorgangs und zweitens die 1000fache Wiederholung dieses Vorgangs.

`InitBenchmark`, `RecBenchmark`

sind nur als Ergänzung zu `PolyBenchmark` gedacht (vgl. die Auswertung in Abschnitt 3.5). Sie berechnen keinen Algorithmus, sondern testen nur das Verhalten der JVM beim Erzeugen von Objekten (`InitBenchmark`) und dem rekursiven Abstieg in einer Funktion (`RecBenchmark`).

3.3 Überlegungen zur Programmiersprache Java

Bei den Testprogrammen wurde Wert darauf gelegt, die von Java und der JVM vorgegebenen Optimierungsmöglichkeiten weitmöglichst auszuschöpfen. Im Folgenden werden die wichtigsten Überlegungen kurz beschrieben:

Verwendung von `final`

Damit der Compiler einen besseren Code erzeugen kann, sollte der Programmierer möglichst immer Methoden, die nicht mehr überschrieben werden, mit dem Modifikator `final` versehen. Z.B.:

```
public final Poly mult(Poly b) { /* */ }
```

aus der Klasse `Matrix`. `private` und `static` Methoden sind implizit `final`.

Vermeidung von `synchronized`

Ein großer Vorteil Javas bei nebenläufiger Programmierung (d.h. bei Verwendung mehrerer Threads) ist ein einfaches Synchronisierungs-Konzept. Kritische Stellen in parallelen Programmteilen, wie gemeinsame Datenstrukturen und deren Methoden können synchronisiert werden. Dies bringt aber auch einen großen Interpretations-Overhead mit sich. Beispielsweise werden so die `java.io` Klassen für zeitkritische Anwendungen schlecht brauchbar. Derzeit sind alle JVM bei Ein/Ausgaben langsam.

Garbage Collection

In Java gibt es keine explizite Speicherfreigabe durch den Programmierer. Diese Aufgabe übernimmt ein System-Thread im Hintergrund. Wenn ein Objekt nicht mehr referenziert wird, gibt er dessen Speicher auf dem Heap wieder frei.

Um Inkonsistenzen in der Zeitmessung zu vermeiden wird in den Benchmarks `System.gc()` verwendet. Dies veranlaßt den Java Interpreter die Garbage Collection gleichzeitig, und nicht nur bei Speicher-Bedarf, durchzuführen.

Um den zusätzlichen Aufwand der Garbage Collection zu vermeiden, kann es sinnvoll sein, häufig erzeugte und verworfene Objekte mit einer eigenen Speicherverwaltung zu versehen. Der Programmierer kann bei Freigabe die Objekte in eine Freiliste aufnehmen, und wenn ein Objekt entsprechender Größe neu angefordert wird, kann so die Erzeugungszeit gespart werden.

In der JVM-Spezifikation ist nicht festgelegt, welchen Algorithmus die Garbage Collection verwendet. Einen ersten Überblick der verschiedenen Techniken bietet der Onlineartikel *Java's garbage-collected heap* [Venn96].

Stärkereduktion

Eine übliche Unterstützung für den Compiler ist die Verwendung von schwächeren äquivalenten Ausdrücken.

Der langsame Zugriff auf Felder in Java kann bei zweidimensionalen Feldern dadurch abgeschwächt werden, daß eine zusätzliche eindimensionale Referenz eingeführt wird. Im folgenden Programmbeispiel aus `MatrixBenchmark` werden die variablen `elem_i`, `elem_max`, `elem_j` statt `elem[i]`, `elem[max]` und `elem[j]` immer genau dann verwendet, wenn in einer Zeile der Matrix mehrere Elemente angesprochen werden.

```
public final void eleminate() {
  double[] elem_i, elem_max, elem_j;
  double dummy;
  int i, j, k, max;
  for (i = 0; i < rows; i++) {
    max = i;
    for (j = i+1; j < rows; j++)
      if (Math.abs(elem[j][i]) >
              Math.abs(elem[max][i])) max = j;
    elem_i = elem[i];
    elem_max = elem[max];
    for (k = i; k < cols; k++) {
      dummy = elem_i[k];
      elem_i[k] = elem_max[k];
      elem_max[k] = dummy;
    }
    for (j = i+1; j < rows; j++) {
      elem_j = elem[j];
      for (k = cols-1; k >= i; k--)
        elem_j[k] -= elem_i[k] *
                          elem_j[i] / elem_i[i];
    }
  }
}
```

Zweidimensionale Felder werden in Java wie eindimesionale Felder von Feld-Objekten behandelt. Jeder Zugriff `elem[i][k]` bedeutet also, daß erst die Referenz auf `a[i]` mit einem `getfield`-Befehl der JVM (vgl. [Sun95b]) auf den Stack geholt wird, und anschließend mit dieser Adresse und einem weiteren `getfield` der Wert der Zelle. Durch Einführen einer zusätzlichen Referenz `elem_i` wird dies umgangen.

Es wird bei jeder Referenz ein `iload`- und ein `aaload`-Befehl im Java-Bytecode eingespart.[2]

Ähnliche Code-Verbesserungen, wie sie in vielen Hochsprachen üblich sind (die Entfernung von invarianten Ausdrücken in Schleifen, die Vorausberechnung von Abbruchbedingungen (`rows` statt `elem.length`) oder die Verwendung der `x += y` Operation statt `x = x + y`) wurden ebenfalls beachtet. Bei einfachen Integer-Variablen optimiert zwar auch der Compiler der JDK (`javac`, mit Parameter `-O`), aber spätestens bei Array-Zugriffen wie `elem_j[k] = elem_j[k] - y` geschieht dies nicht mehr. So verbesserter Code ermöglicht auch JITC weitergehende Optimierungen.

`System.arraycopy()`

Eine weitere Möglichkeit der Performancesteigerung im Java-Code ist die Verwendung der *richtigen* Java-Methode. Zum Beispiel bietet Java eine schnelle System-Funktion für das (vollständige oder teilweise) Kopieren von Arrays an. Mit `java.lang.System.arraycopy()` im folgenden Code-Abschnitt:

```
Poly al = new Poly(len2);
Poly bl = new Poly(len2);

try {
  System.arraycopy(elem, 0, al.elem, 0, len2);
  System.arraycopy(b.elem, 0, bl.elem, 0, len2);
}
catch(ArrayIndexOutOfBoundsException e) { }
catch(ArrayStoreException e) { }
```

statt einer einfachen Schleife:

```
for (int i = 0; i < len2; i++) {
  al.elem[i] = elem[i];
  bl.elem[i] = b.elem[i];
}
```

kann so Ausführungszeit in kritischen Bereichen eingespart werden.

Profiling

Um diese zeitkritischen Bereiche zu finden, stellt der Interpreter (der JDK) die Möglichkeit des Profiling zur Verfügung. Profiler sind Hilfprogramme, die messen, wieviel Laufzeit die einzelnen Teile eines Programms verbrauchen und wie oft Methoden aufgerufen werden. Den Java-Profiler wird mit dem Befehl `java -prof <Klasse>` aufgerufen und anhand der in `./java.prof` abgelegten Statistik, können laufzeitfintensive Programmteile erkannt werden. Die in Anhang A angefügten Benchmark-Programme sind so auf ihre Zuverlässigkeit getestet.

[2] http://www.cs.cmu.edu/~jch/java/compilers.html

3.4 Test-Bedingungen

3.4.1 Die drei Testplattformen

Im Folgenden sind die wichtigsten technischen Daten der drei Tesplattformen tabellarisch aufgelistet:

SPARCstation 4

Modell:	SPARCstation 4
Hersteller:	Sun Microsystems
RAM:	48 MB
Virtueller Speicher:	141 MB
CPU Anzahl:	1
CPU Typ:	85 MHz microSPARC II
Kernel Architektur:	sun4m
Betriebssystem:	SunOS 5.4 (Solaris)
Virtueller Adress Cache:	16384
Software:	JDK 1.0.2[3] JDK 1.0.2dp11/13/96-15:42 (neuere Version) Netscape Navigator 3.01[4] kaffe 0.7.0[5]

PC - Windows 95

CPU:	Pentium 90
RAM:	48 MB
Cache:	256 KB
Betriebsystem:	Microsoft Windows 95
Software:	JDK 1.0.2ss:08/12/96-14:32:12 (neuere Version[6]) Symantec Visual Cafe prebeta II[7] Asymetrix SuperCede 1.0 beta[8]

[3] http://www.javasoft.com/
[4] http://www.netscape.com/
[5] http://www.sarc.city.ac.uk/˜tim/kaffe/index.html
[6] Teil des *Java Workshops für Windows* von Sun Microsystems
[7] http://www.symantec.com/
[8] http://www.asymetrix.com/

Microsoft Internet Explorer 3.01[9]
Netscape Navigator 3.02 (mit JIT von Borland)

PC - Linux 2.0.x

Hardware:	vgl. PC - Windows 95
Betriebssystem:	Linux 2.0.x mit XFree86 3.1.2
Software:	JDK 1.0.2chapman:10/12/12-23:12[10] Netscape Navigator 3.01 Kaffe 0.7.0

3.4.2 Sonstige Festlegungen

Alle Quelltexte wurden mit dem Java-Compiler `javac` von Sun Microsystems mit der Optimieroption `-O` in Bytecode übersetzt.

Als Referenz-JVM ist auf jeder der Plattformen die JDK 1.0.2 von JavaSoft (bzw. bei Linux der Port von RANDY CHAPMAN) installiert. Deren Ergebnisse werden mit 100.0% gewertet, und abhängig davon der Prozentwert der anderen Interpreter/JIT-Übersetzer[11] berechnet. So ist ein plattformübergreifer Vergleich an Relativwerten möglich.

Die Tests sind so dimensioniert, daß die Ausführungszeit mit dem Interpreter der JDK 1.0.2 zwischen einer und drei Minuten liegt.

3.5 Ergebnisse

Im Anhang B sind die genauen Meßwerte der Testläufe tabellarisch für die einzelnen Plattformen aufgeführt. Abhängig von Interpreter und Testprogramm sind je vier Zeitmessungen vorgenommen worden. Um den Einfluß von zufälligen Systemereignissen zu unterdrücken, wurde der größte und der kleinste Wert für den weiteren Vergleich ignoriert. Die jeweils fünfte Zeile ist der Durchschnittswert der beiden gewerteten Zahlen. Für ihn ist auch der Relativwert berechnet. Die Abbildung 3.3 zeigt ein Beispiel für eine Meßreihe.

SPARCstation 4 - Solaris

Neben Windows NT war Solaris (Abbildung 3.4) die erste Plattformen, für die Sun seine Referenzimplementierung einer JVM, die JDK, angeboten hat. Getestet wurden zwei verschiedene Versionen, die normale Distribution und eine spätere Version: dp11/13/96-15:42. Außer bei den Benchmarks mit Matrix- und Polynommultiplikation (105%) bringt diese bessere Ergebnisse. Die verschiedene Versionen der JDK 1.0.2 ergeben sich aus verschieden Freigabeterminen bei Sun/Javasoft. Mit neuen Entwicklungen wie dem Browser *HotJava* oder dem *Java-Workshops*, einer grafischen Entwicklungsumgebung für Java-Programme, ist auch immer

[9] http://www.microsoft.com/ie/
[10] http://www.blackdown.org/
[11] Das Konzept der Just in Time-Compiler wird in 4.2 näher beschreiben.

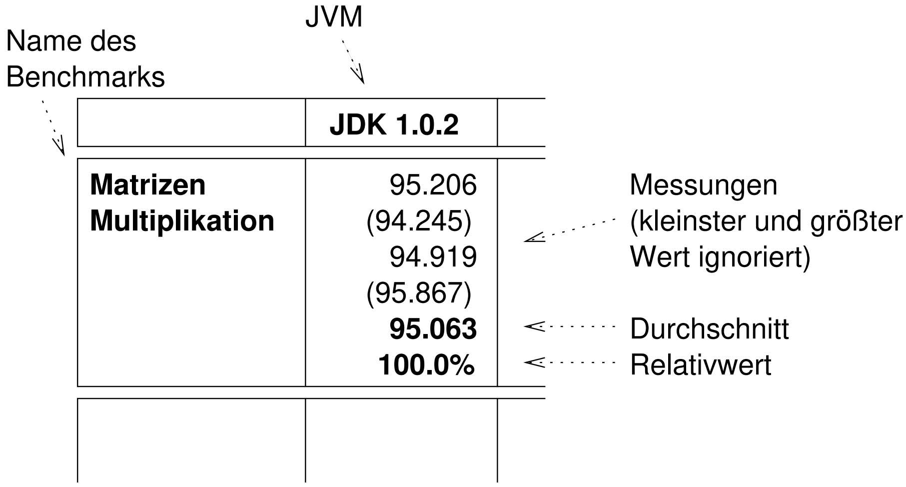

Abbildung 3.3: Berechnungsbeispiel der Ergebnisse

die entsprechend aktuellste Version des Interpreters erhältlich. Insbesondere Fehler früherer Versionen[12] sind dort behoben.

Kaffe ist eine frei erhältliche JVM, die für mehr als 20 Hardware/Betriebssystem-Varianten erhältlich ist. Sie ist eine sogenannte Clean-Room-Implementierung, d.h. sie baut nicht auf den Code von Sun Microsystems auf. Auf der Intel Plattform und — seit der hier getesteten Version 0.7.0 — auch auf SPARC unterstützt sie eine Just in Time-Übersetzung in maschinenabhängigen Code. Und obwohl sie immernoch ALPHA-Software ist, ist sie schon relativ stabil und sicher. Kaffe beinhaltet keine AWT, d.h. es wird keine Native-Library für das Abtract Window Toolkit mitentwickelt. Für Unix/X ist es aber möglich die in 5.1.4 beschriebenen AWT-Implementierungen einzubinden.

Im Test zeigt kaffe 0.7.0 (ausgenommen von `PolyBenchmark`) eine Laufzeitverbesserung von im Schnitt Faktor 5. Bei Aufgaben wie dem Primzahlensieb oder dem mehrfachen Sortiern der Integerliste ist die Steigerungen deutlicher. Das Lösen der Gaußelimination gelingt immerhin noch in 35.8% der Zeit, die der reine Interpretierer von Sun Microsystems benötigt. Negativ fällt nur der Test mit der Polynommultiplikation auf, mit 235.3% ist kaffe hier mehr als doppelt so langsam. Gründe hierfür werden im nächsten Abschnitt bei den JIT-Compilern für Windows gesucht.

Der Netscape Navigator 3.01 ist unter Solaris neben dem HotJava von Sun, der einzige WWW-Browser, der eine Java-Unterstützung bietet. Da HotJava direkt auf die JDK aufsetzt wurde auf einen Test verzichtet. Der Browser von Netscape unterstützt auf den Unix-Plattformen keine JIT-Compilierung, deshalb sind seine Ergebnisse bei den Benchmarks auch nicht weiter verwunderlich. Nur der Polynomtest und das Erzeugen des Primzahlenobjekts ergaben Abweichungen nach oben von bis zu 30 Prozentpunkten. Die restlichen Werte lagen zwischen 101.0% und 117.0%.

[12] http://www.javasoft.com/products/jdk/1.0.2/KnownBugs.html

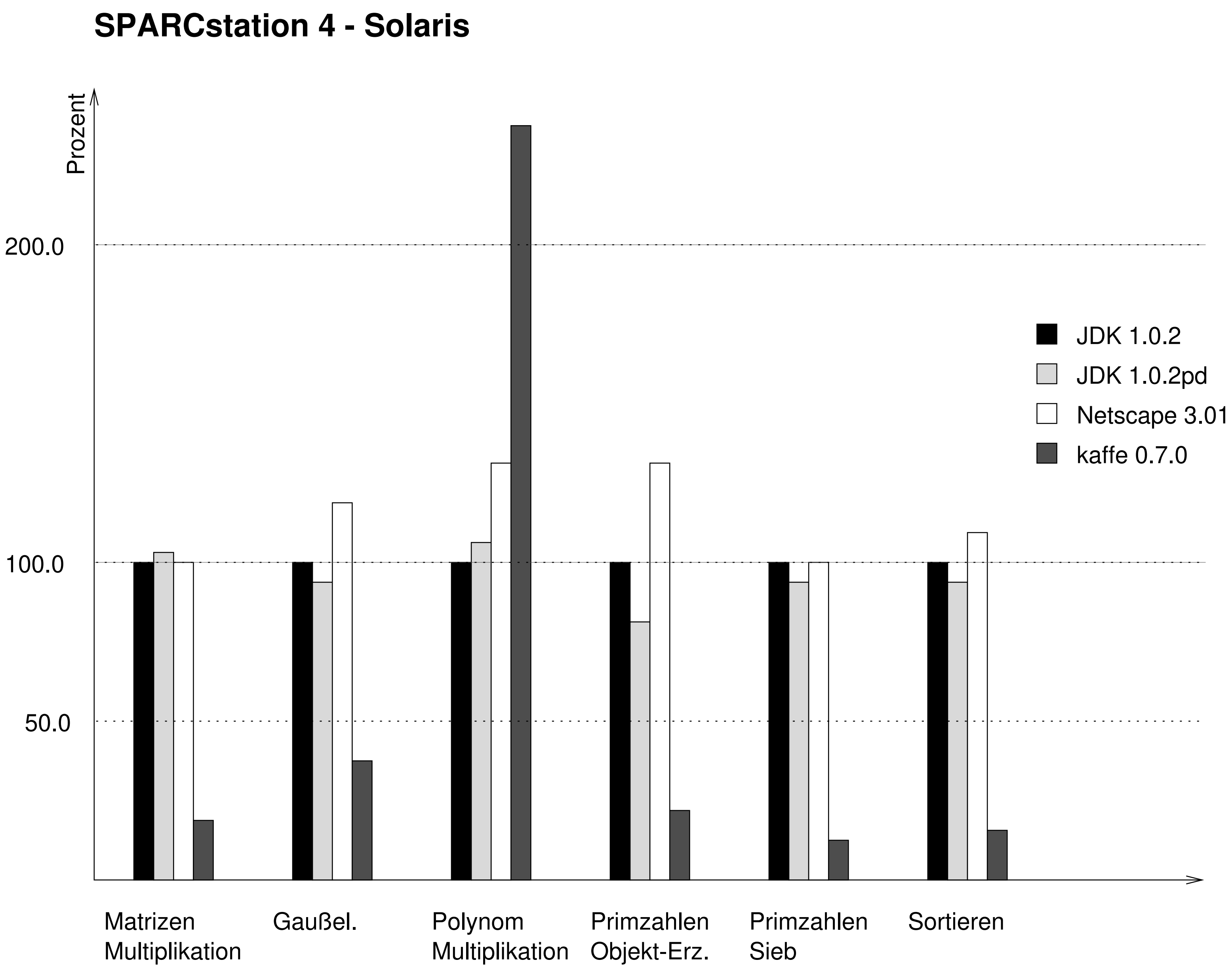

Abbildung 3.4: Ergebnisse SPARCstation 4 - Solaris, alle Benchmarks

Pentium 90 - Windows 95

Besonders für Windows[13] bieten viele Softwarehersteller JIT-Compiler (JITC) an. Bis auf die JDK 1.0.2 besitzen alle unter Windows getestete JVM diese Option. Im Überblick (vgl. Abbildung 3.5) bestätigen dies auch die Zahlen der Benchmarks. Bis zu Faktor 20 kann hier der Unterschied betragen (Primzahlensieb). Bei der Lösung des Gleichungssystems mit 400 Unbekannten und 400 Gleichungen liegen die vier JITC-Ergebnisse alle in einem Bereich von 10.2-12.4% des JDK-Wertes. Diese gleichen Leistungswerte sind aber keineswegs üblich, nur bei Berechnung des Primzahlensiebs haben die Werte eine ähnlich kleine Streuung (5.2-7.7%).

Grundsätzlich liegt die Ausführzeit der JVM im Netscape Browser in einer Größenordnung von 0.5 bis 1 über denen des Browsers von Microsoft (Internet Explorer). Auch die beiden Stand-alone-Produkte von Asymetrix und Symantec zeigen deutliche Unterschiede. Bei Matrizenmultipikation, Polynommultiplikation und im Sortieren der Integerliste übertrifft Visual Cafe PreBeta II SuperCede von Asymetrix in einem Faktor zwischen 1.8 und 4.2. Bei den Benchmarks mit Gaußelimination, Erzeugen des Objekts für den Primzahlentest und dem Primzahlensieb-Algorithmus selbst sind die Zeitwerte von SuperCede besser. Bei Prozentwerten von 5 bis 10 liegen die Abweichungen hier aber nur bei 2 bis 3 Prozentpunkten.

Auffallend aber sind besonders die Ergebnisse des Benchmarks mit der Multiplikation zweier Polynome der Länge 4000. Der Netscape Navigator hat hier eine doppelt so langen Ausführungszeit (203.7%) wie das Referenzsystem, die JDK von Sun Microsystems. Auch die anderen JITC ergeben hier keine ähnlichen Geschwindigkeitsverbesserungen wie bei den anderen Tests. SuperCede ist mit einem Wert von 130.2% langsamer, genau wie der Internet Explorer (IE) mit 108.1% und nur Symantec Visual Cafe ist mit 46.2% schneller als der reine Interpretierer, der JDK.

`PolyBenchmark` implementiert wie in Abschnitt 3.2 beschrieben eine rekursive Polynommultiplikation, in der das Problem in drei Teilprobleme der halben Größe geteilt wird und danach die Ergebnisse wieder zusammengesetzt werden. Hierfür werden in jedem Rekursionsschritt sechs neue `Poly`-Objekte erzeugt.

Dies stellt sich für die JITC als großes Problem dar. Um zu entscheiden, welche Programmteile die Beschleunigung durch die Just in Time-Compiler behindern, wird der Test in zwei aufgeteilt: `InitBenchmark` und `RecBenchmark`. Die Ergebnisse von `InitBenchmark` und `RecBenchmark` (vgl. für den Programmcode Anhang A) in Tabelle 3.1 zeigen, daß SuperCede, der Netscape Navigator und der Internet Explorer signifikant mehr Zeit bei der Erzeugung von neuen Objekten brauchen, als die JVM von JavaSoft. Daß die schlechten Resultate bei `PolyBenchmark` nicht von der Rekursion im Teile-und-Herrsche-Algorithmus abhängt zeigt auch `RecBenchmark`, ein Test, der eine einfachen rekursiven Abstieg in *einem* Objekt beinhaltet (vgl. Tabelle 3.1). Der Großteil der Ausführungszeit wird deshalb für das Erzeugen der Objekte aufgewendet.

Bei `InitBenchmark` braucht SuperCede ca. 2,8 mal länger (281.3%) als der Interpretierer der JDK von JavaSoft. Der nicht ebenfalls so schlechte Wert bei `PolyBenchmark` resultiert deshalb aus der guten Optimierung der restlichen Programmteile.

Visual Cafe von Symantec ist in der Lage, bei `InitBenchmark` die Zeit der JDK zu

[13] Die Tests wurden unter Microsoft Windows 95 vorgenommen, da mit Microsoft Windows NT nur ein 486/33 zur Verfügung stand und besonders bei den JIT-Compilern die Leistung der Hardware wichtig wird. So ist auch ein besserer Vergleich mit den Ergebnissen unter Linux auf der gleichen Hardware möglich.

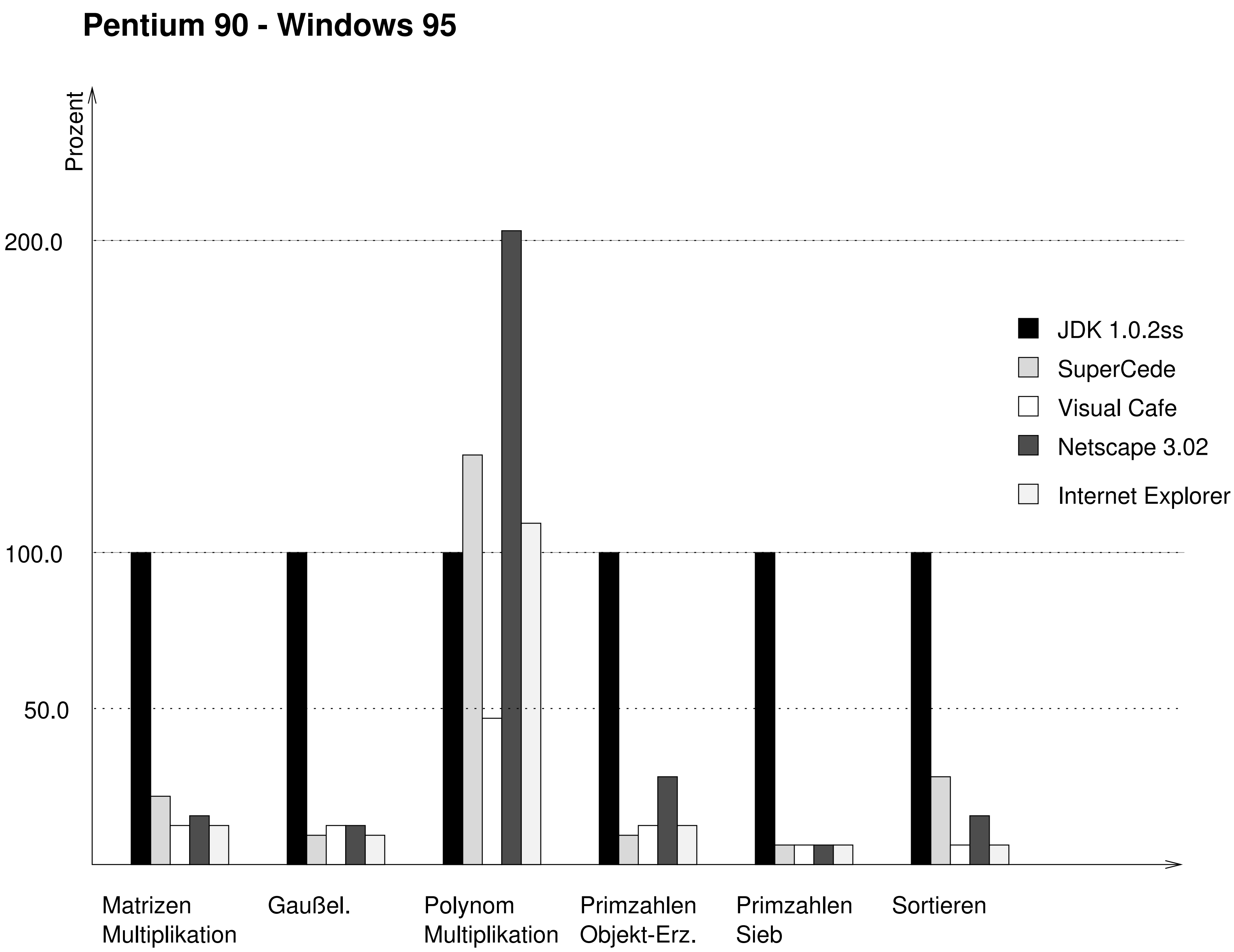

Abbildung 3.5: Ergebnisse Pentium 90 - Windows 95, alle Benchmarks

	JDK 1.0.2 ss08/12/96	Super-Cede	Visual Cafe 2.00b07	Netscape 3.01	IExplorer 3.01
Erzeugen von	(52.68)	(158.41)	51.57		(72.78)
Objekten	(52.89)	151.65	(51.63)	länger	72.12
	52.73	145.06	(51.36)	als 8 Min.	71.96
	52.73	(144.84)	51.61		(71.30)
	52.73	**148.36**	**51.59**		**72.04**
	100.0%	**281.3%**	**97.8%**		**136.6%**
Rekursiver	**12.25**	**1.37**	**0.88**	**0.94**	**0.99**
Abstieg	**100.0%**	**11.2%**	**7.2%**	**7.7%**	**8.1%**

Tabelle 3.1: Ergebnisse für `InitBenchmark` und `RecBenchmarks` unter Windows 95

erreichen und knapp zu unterbieten. Auch hier wird der Zeitvorteil in `PolyBenchmark` durch Optimierung der anderen Programmkonstrukte erreicht.

Für die Programmentwicklung hat dies konkrete Folgen: Um nicht Performanceeinbußen hinnehmen zu müssen, ist das allzu häufige Erzeugen von Objekten zu vermeiden. In `PolyBenchmark` könnte dies zum Beispiel mit der Einführung einer Freiliste für nicht mehr verwendete `Poly`-Objekte und deren Recycling bei nächster Gelegenheit erreicht werden. Da die Rekursion einen Tiefensuchebaum erzeugt, d.h. einen Teilzweig immer erst bis zum letzten Blatt abarbeitet, könnten die auf diesem Weg erzeugten Objekte im nächsten Teilzweig wiederverwendet werden. Damit wird die Zeit zur Erzeugung eingespart.

Ein positiver Nebeneffekt wäre die Umgehung der Garbage Collection von Java. Laut Berichten in Newsgroups[14] und verschiedener Benchmarks[15] ist sie bei den meisten Interpretern und JIT-Übersetzern nur schlecht implementiert.

Pentium 90 - Linux

Unter Linux (vgl. Abbildung 3.6) standen genau wie bei Solaris nur die JDK 1.0.2 (nicht in einer offiziellen Portierung von Sun), der Netscape Navigator 3.02 (ohne JITC) und kaffe 0.7.0 als Testsysteme zur Verfügung.

Die Ergebnisse des Netscape Navigators sind mit denen der anderen Unix-Plattform vergleichbar. Die Primzahlentests sind mit Prozentwerten von 81.2% und 83.2% besser als die Werte des Interpreters von JavaSoft; die Multiplikation der Polynome braucht 154.5% der Ausführungszeit der JDK.

Auch kaffe ergibt fast die gleichen Werte wie bei der SPARC-Version. Da die JITC-Option hier schon länger eingebaut und getestet ist unterstreicht das die Leistung bei der Solaris-Version, die erst mit der Version 0.7.0 eine Just in Time-Compilierung ermöglicht.

Das Erzeugen des großen Arrays für das Primzahlensieb ist unter Linux genau wie der Primzahlensieb-Algorithmus selbst und `SortBenchmark` relativ zur JDK doppelt so schnell wie unter Solaris. Ebenfalls negativ fiel PolyBenchmark auf. Die Werte lagen über dem Doppeltem der Sun-Ergebnisse. Wärend der Testläufe wurde eine Ausführung mit einer

[14] de.comp.lang.java und comp.lang.java.*

[15] http://www.cs.cmu.edu/~jch/java/benchmarks.html

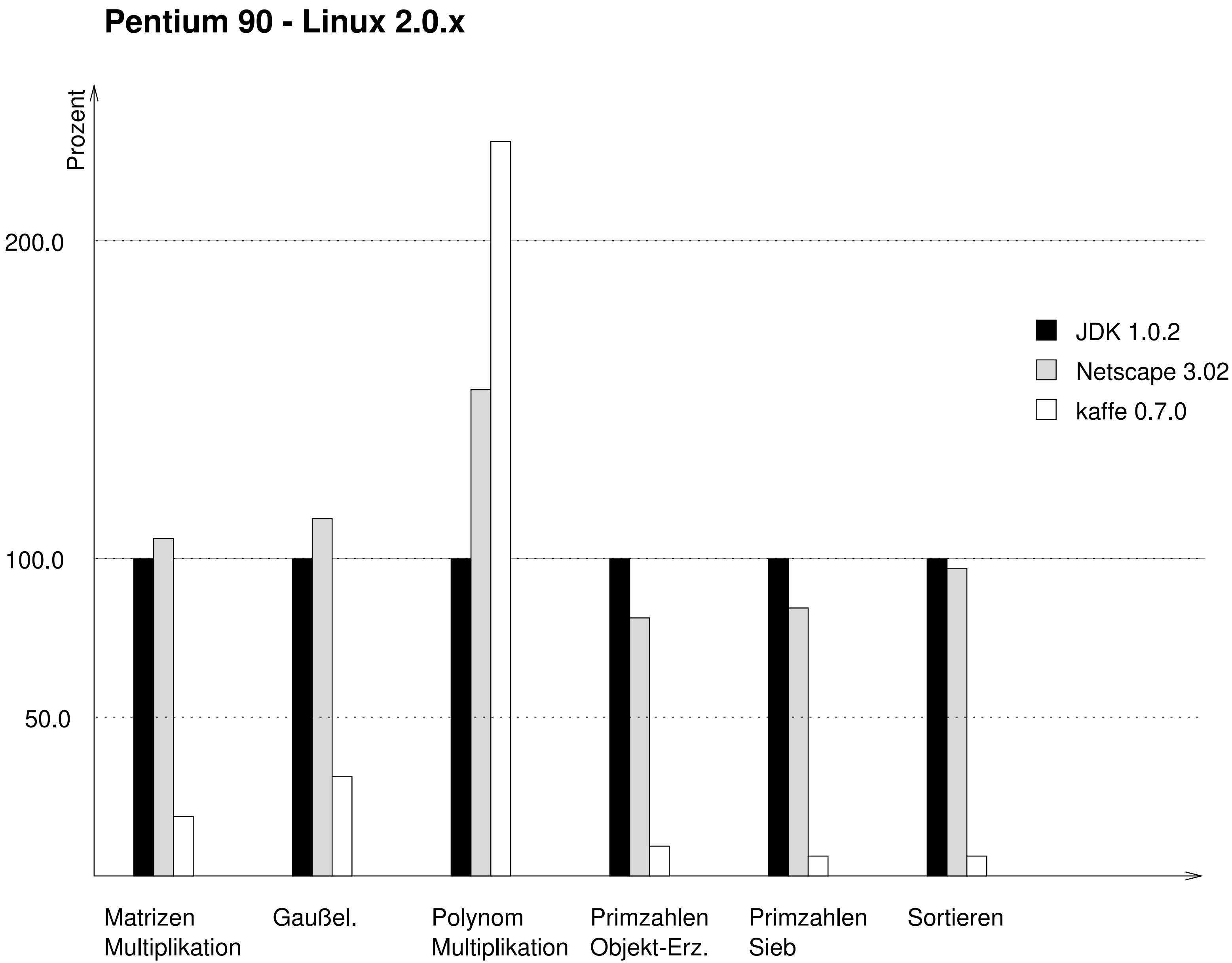

Abbildung 3.6: Ergebnisse Pentium 90 - Linux, alle Benchmarks

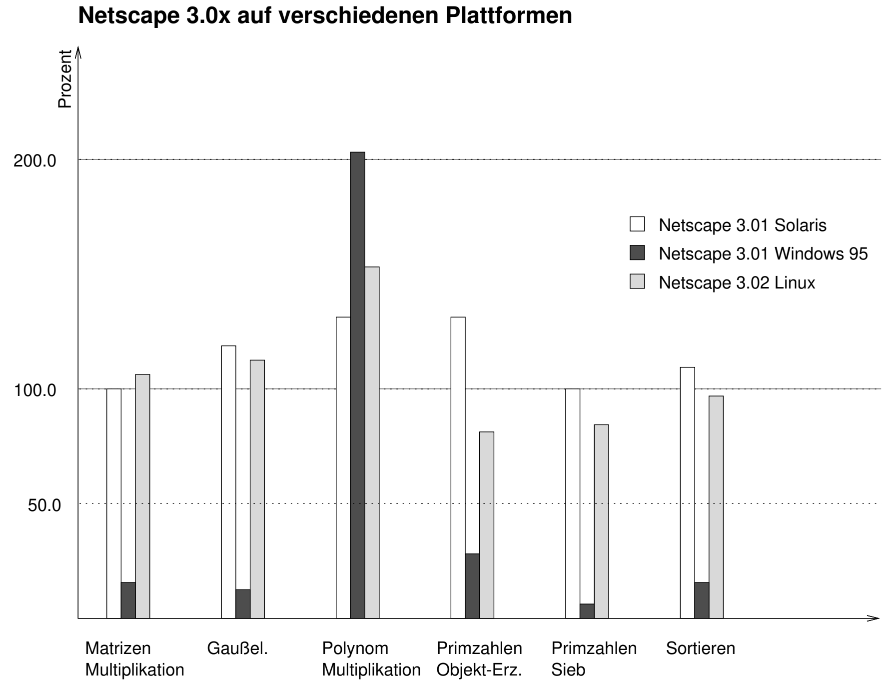

Abbildung 3.7: Ergebnisse mit Netscape auf allen Plattformen, alle Benchmarks

`java.lang.NullPointerException` beendet. Dieser Fehler ließ sich nicht wiederholen und deutet auf einen internen Fehler der kaffe JVM hin.

Allgemein

Die Tests für Windows 95 und Linux/X wurden auf der gleichen Hardware ausgeführt. Deshalb lassen sich hier bedingt auch die absoluten Zahlen vergleichen. Deutlich zu erkennen ist, daß der Netscape Navigator unter Windows über eine JITC verfügt (vgl. Abbildung 3.7). Über die Betriebssystemgrenzen bewirkt dies eine Steigerung bis zu einem Faktor von 20.

Natürlich unterscheiden sich auch die Leistungswerte der JDK selbst. Hier schneidet die Windows-Version um durchschnittlich 30% besser ab (vgl. Abbildung 3.8).

3.6 Ergebnisse von CaffeineMark

Für den *normalen* Java-Benutzer geschieht die Benutzung derzeit oft nur als Applet in einem Browser. Pendragon Software[16] versucht mit einem Benchmark-Paket, dem CaffeineMark, die Leistungsfähigkeit der eingebauten JVM der Browser von Microsoft (IE), Netscape (Navigator) und Sun (HotJava, ohne JITC) zu vergleichen.

Der CoffeinMark 2.5 besteht aus 9 Tests die alle Bereiche der Java-Programmierung abdecken. Zusätzlich zu numerischen Berechnungen, wie sie Grundlage der Benchmarks dieser

[16] *Battle of Browsers*: http://www.webfayre.com/battle.html

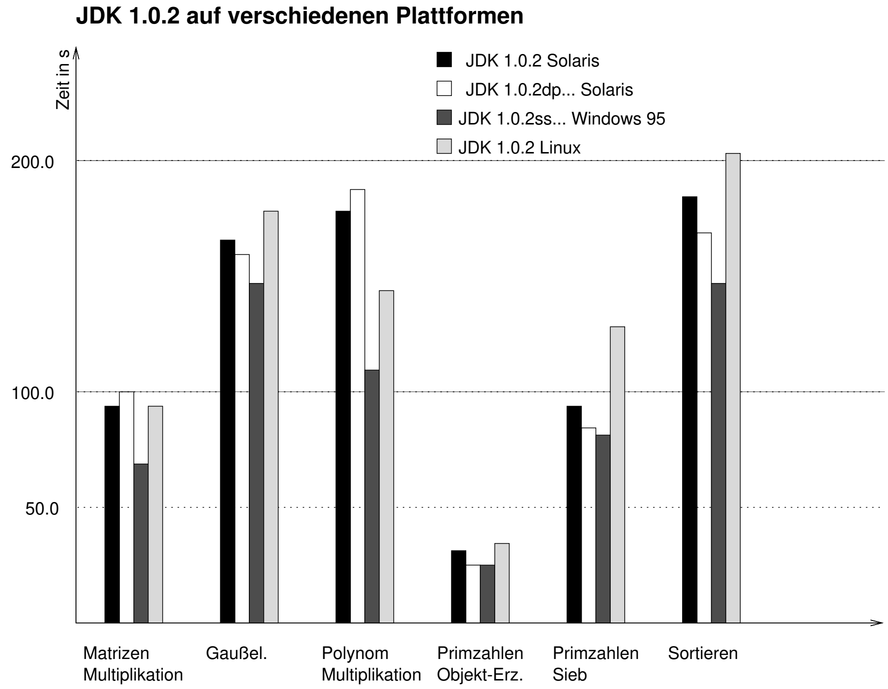

Abbildung 3.8: Ergebnisse mit JDK 1.0.2 auf allen Plattformen, alle Benchmarks

Studienarbeit sind, werden auch Performace-Aspekte der Grafik und Zeichenkettenverarbeitung beachtet. Der CoffeineMark ist ein Mittelwert aus den 9 Test-Ergebnissen und gibt relativ zu einem Referenzsystem (mit Cyrix 6x86 P150+, Windows 95 und dem Symantec Cafe appletviewer im Debug-Modus) einen Leistungswert für die getestete JVM.

Die Abbildung 3.9 zeigt die Ergebnisse für HotJava PreBeta 1, Internet Explorer 3.0B1 und Navigator 2.02. Diese Browser-Versionen verfügen noch nicht über eine JIT-Compilierung.

Im Vergleich dazu zeigen die beiden Browser Internet Explorer 3.0B2 und Navigator 3.0B5A in Abbildung 3.10 deutliche Steigerungen der Ausführungsgeschwindigkeit. Besonders „sieve, loop, logic floating-point (fp) und der method Test“, so Pendragon, „werden dramatisch bescheunigt“. Dies kommt besonders „berechnungsintensive Animationen und 3D-Modellierung“ (Pendragon) zugute.

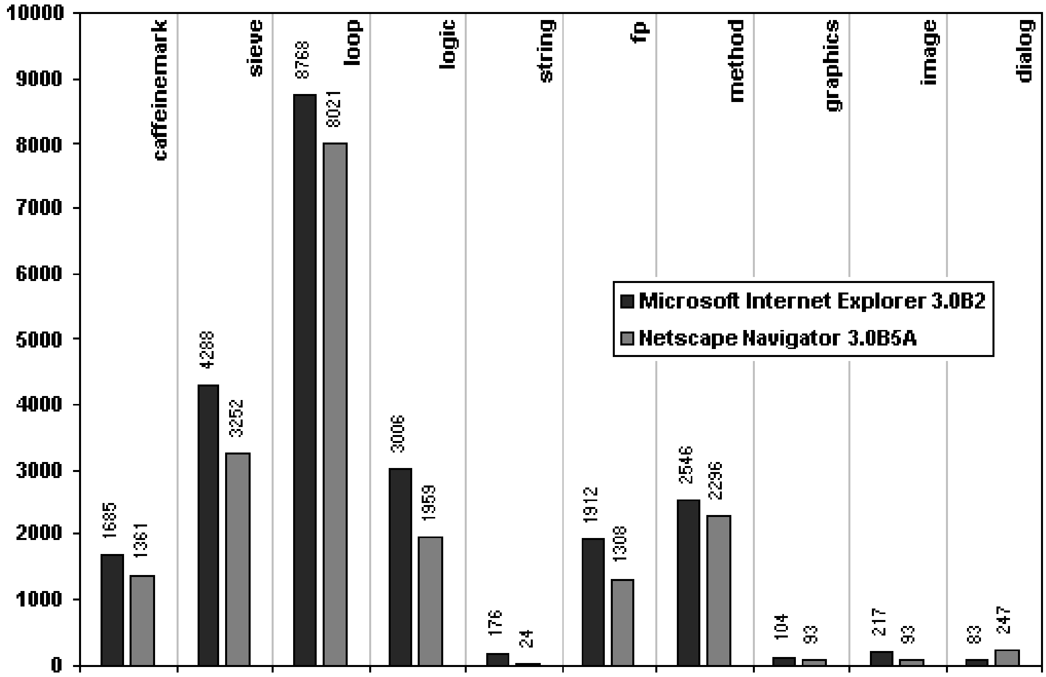

Abbildung 3.9: Ergebnisse von CaffeineMark 2.5: JVM ohne JITC

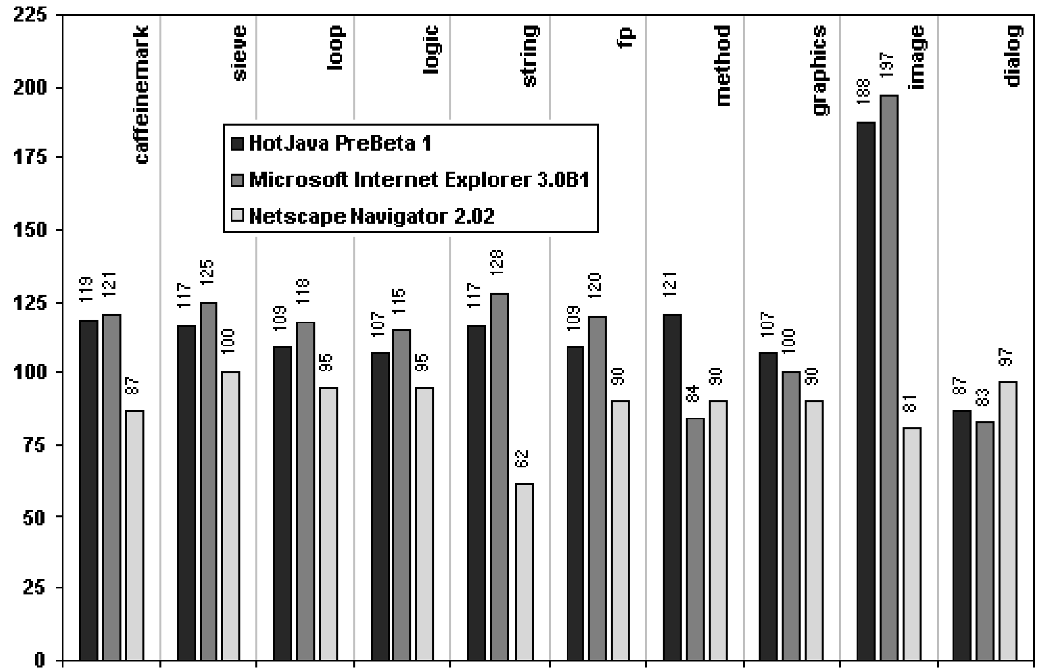

Abbildung 3.10: Ergebnisse von CaffeineMark 2.5: JITC

Kapitel 4

Laufzeitverbesserungen

4.1 Natives

Nach der Untersuchung des wirklichen Laufzeitverhaltens von Java Programmen und JVM liegt es nahe, Konzepte der Laufzeitverbesserung zu behandeln.

In `Benchmark` (vgl. Abschnitt 3.3) wird `System.arraycopy()` verwendet, eine für das Kopieren von Arrays optimierte Routine der JVM.

Was für das Kopieren von Arrays schon vorgesehen ist, kann auch für andere Probleme selbst implementiert werden. Java sieht `native` Methoden vor. D.h. es lassen sich aus dem Java-Code heraus direkt C (C++)-Routinen aufrufen. JavaSoft hat für diesen Fall entsprechende Tools und C-Include-Files vorgesehen. Sie sind Teil der Standard-Distribution und ermöglichen die Kommunikation zwischen C- und Java-Ebene. C++-Routinen können ebenfalls über diese Schnittstelle eingebunden werden [Foot96].

Durch diese Änderung auf Ebene des Java-Bytecodes (vgl. Abbildung 4.1) geht zwar die Systemunabhängigkeit verloren, es kann aber bei rechenintensiven Beispielen deutliche Geschwindigkeitsvorteile bringen. Außerdem besteht die Möglichkeit, zusätzlich zu den systemabhängigen Methoden langsamere Äquivalente in Java zu schreiben, die ausgeführt werden, wenn die dynamisch ladbare Bibliothek nicht zur Verfügung steht (vgl. das nächste Beispiel), z.B. auf einer anderen Hardwareplattform.

Die folgende Klasse `UseNative` stellt die Einbindung von `native` Methoden vor. Die Methoden `displayHelloWorld()`, `add_ab1()` und `add_ab2()` sind außerhalb des Java Codes implementiert. Verwendet werden können sie aber genauso wie normale Methoden:

```
UseNative use = new UseNative();
```

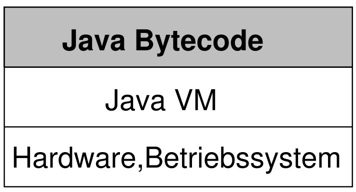

Abbildung 4.1: Abstrakte Ebenen von Java: Java Bytecode

```
use.displayHelloWorld();
```

Beispielsweise gibt `use.displayHelloWorld()` *Hello World* an der Standardausgabe aus.

Die Methode `displayHelloWorldFallBackCode()` erfüllt die gleiche Aufgabe, prüft aber vorher ob der Ladebefehl für die Shared Library funktioniert hat und führt andernfalls eine entsprechende Java Routine aus.

Auch Parameter können der `native` Methode mitübergeben werden. Die Funtionen

```
use.add_ab1(use.a, use.b);
use.add_ab2(use.b);
```

geben beide den Additionswert der Variablen `a` und `b` zurück. Im ersten Fall werden beide als Paramter an die native Routine übergeben, im zweiten nur `b`. Auf `a` greift die C-Routine über die von der JDK vorgesehene Schnittstelle zu.

Im Folgenden wird der Beispiel-Code für die Klass `UseNative` gegeben:

```
class UseNative {
  private static boolean nativeCode;

  public native void displayHelloWorld();
  public native int add_ab1(int a, int b);
  public native int add_ab2(int b);

  public void displayHelloWorldFallBackCode() {
    if (nativeCode)
      displayHelloWorld();
    else
      System.out.println("Hello World");
  }

  static {
    try {
      System.loadLibrary("hello");
      nativeCode = true;
    }
    catch (UnsatisfiedLinkError e) {
      System.out.println("Library not found.");
      nativeCode = false;
    }
    catch (Exception e) {
      System.out.println("Cannot load Library.");
      nativeCode = false;
    }
  }

  public int a;
  public int b;
```

```
}
```

Der dazugehörende C Code der Datei `MyUseNative.c` sieht so aus:

```
#include <StubPreamble.h>
#include "UseNative.h"
#include <stdio.h>

/* Ausgabe an stdout */

void UseNative_displayHelloWorld(struct
                            HUseNative *h) {
  printf("Hello World!\n");
  return;
}

/* Addition der beiden Parameter */

long UseNative_add_ab1(struct HUseNative *h,
                       long x, long y) {
  return x + y;
}

/* Addition der int-Variable a des Java
 * Objekts mit dem übergebenen Paramter
 */

long UseNative_add_ab2(struct HUseNative *h,
                       long y) {
  return unhand(h)->a + y;
}
```

Um dem Java Programm die C Funktionen zur Verfügung zu stellen, sind nur drei Arbeitschritte nötig:

1. Übersetzen des Java Codes:

   ```
   javac UseNative.java
   ```

2. Erzeugen der C-Header und Source Dateien, die die Klasse `UseNative` beschreiben und eine Schnittstelle zwischen Java und C zur Verfügung stellen:

   ```
   javah UseNative
   javah -stubs UseNative
   ```

3. Übersetzen des eigenen C-Codes und Binden zu einer Library
   Unter Solaris geht dies zum Beispiel in einer Befehlszeile:

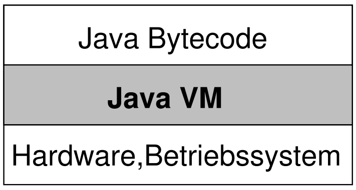

Abbildung 4.2: Abstrakte Ebenen von Java: Java Virtual Machine (JVM)

```
gcc -G MyUseNative.c UseNative.c
    -o libhello.so
    -I/export/opt/java/include
    -I/export/opt/java/include/solaris
    -I/export/opt/GNU/lib/g++-include
```

Nicht nur die Verwendung von Code in C oder C++ zum Geschwindigkeitsgewinn ist für Softwareentwickler interessant. Auch die Wiederverwendbarkeit von *altem* Code ist so möglich.

Mit der Beta-Release der JDK 1.1 hat JavaSoft ein neue genau spezifizierte Schnittstelle für `native` Methoden eingeführt, das *Java Native Method Interface* [Java96b].

4.2 JIT/ Ahead-of-Time-(Re)Compiler

Die Laufzeiteffizienz kann ebenfalls auf der abstrakten Ebene der JVM gesteigert werden (vgl Abbildung 4.2).

Ein Konzept sind sogenannte *Just In Time*-Übersetzer, die parallel zur Ausführung oder dem Laden der Klasse ausgeführt werden. Sie generieren Native-Code *on-the-fly*[1], der schneller ausgeführt werden kann. Der JIT-Code kann dann auch plattformspezifische Hardwarevoraussetzungen, wie einen Coprozessor oder ähnliches, nutzen. Die Abbildung 4.3 verdeutlicht den Unterschied zwischen Interpretation und Just in Time-Übersetzung.

Die JDK von Sun sieht für die Einbindung eines JIT-Compilers die Klasse `java.lang.Compiler` vor [Java96a]. Sie ist sowohl zuständig für das Feststellen ob eine entsprechende Bibliothek mit JIT-Compiler existiert, als auch für die Initialisierung; beides geschieht im static-initializer-Block beim Laden der Klasse:

```
try {
  String library = System.getProperty("java.compiler");
  if (library != null) {
    System.loadLibrary(library);
    initialize();
  }
} catch Throwable e) { }
```

[1] d.h. während der Ausführung, direkt bei Bedarf

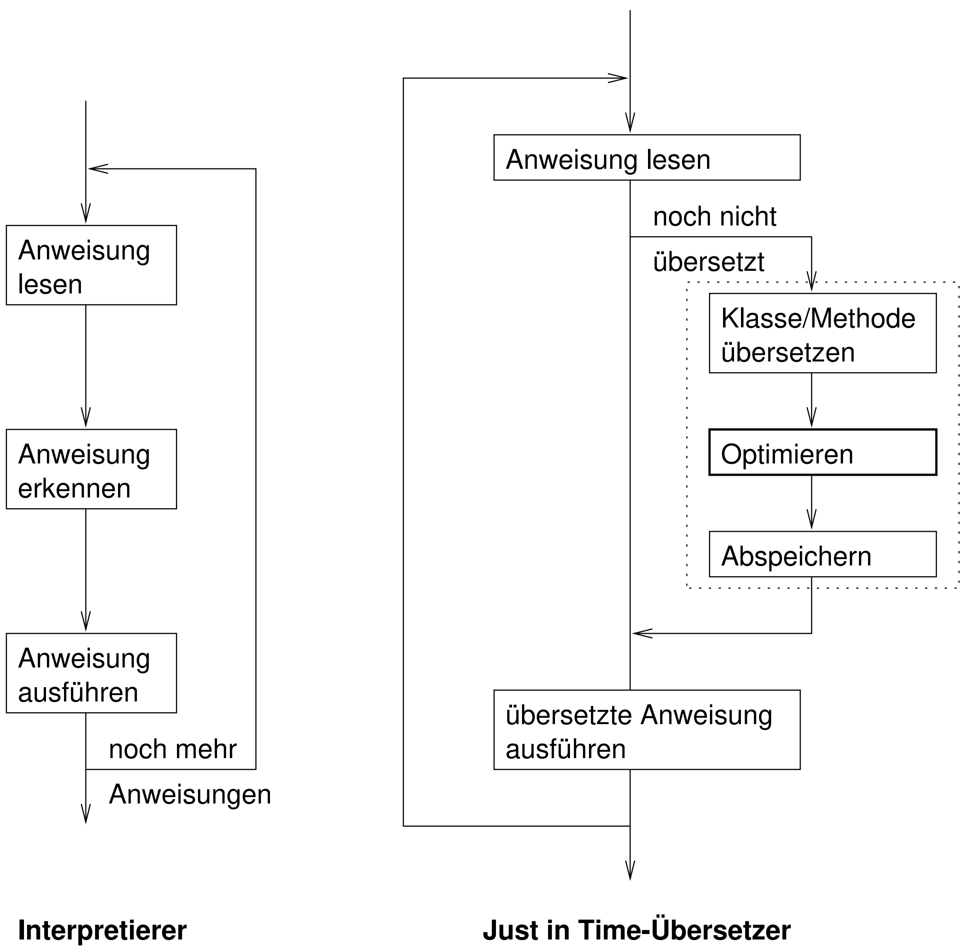

Abbildung 4.3: Interpretation *vs* JITC

Mit der `System`-Routine `getProperty()` wird die Umgebungsvariable `JAVA_COMPILER` gelesen und die dort optional angegebene Bibliothek dynamisch geladen.

Außerdem enthält die Klasse verschiedene Methoden mit denen Applikationen Informationen an den Compiler senden und empfangen können:

```
public static void disable();
public static void enable();
public static boolean compileClass(Class clazz);
public static boolean compileClasses(String string);
public static Object command(Object any);
```

Die genaue Beschreibung dieser Methoden und weiterer Strukturen findet sich in [Java96a].

Vor oder parallel zur Ausführung einer Klasse wird ihr Bytecode in maschinenabhängigen Code übersetzt, der dann nicht mehr interpretiert werden muß, sondern wie ein normales Programm ausgeführt werden kann. Besonders dann, wenn weiterübersetzter Code mehrfach verwendet wird, kann die JIT-Übersetzung die Ausführungszeit reduzieren.

Die oben getesteten JVM von Symantec, Asymetrix, Microsoft und auch die frei erhältliche JVM kaffe verfügen über eine JIT-Übersetzereinheit. In der Windows-Version des Navigators setzt Netscape derzeit noch die JIT-JVM von Borland ein.

Das Übersetzen des Bytecodes in Maschinenbefehle ermöglicht auch die Ausnutzung spezifischer Hardware. Der erzeugte Code kann wesentlich weiter optimiert werden als dies für die Stack-Maschine der JVM möglich ist.

Besonders bei rechenintensiven Programmen sparen JIT-Compiler Zeit und hier ist dann auch mehr Aufwand bei der Optimierung durch kürzere Ausführzeiten leicht aufzuwiegen. Derzeit sind die Produkte auf dem Markt sicher noch nicht am Ende ihrer Entwicklungsmöglichkeiten angelangt, schon der Performanceunterschied der einzelnen JIT-JVM deutet darauf hin.

Was ein JIT-Übersetzer on-the-fly durchführt (vgl. markierter Ablaufpfad in Abbildung 4.3), kann auch vorher von einem *Ahead-of-Time*-Compiler oder Recompiler berechnet und in einem *fat* Classfile abgelegt werden [Java96a]. Fat Classfiles enthalten neben dem normalen Java-Bytecode plattformabhängigen Maschinencode für die gesamte Klasse oder nur für bestimmte Methoden. Es ist auch möglich den Code von mehreren Maschinenarchitekturen hier abzulegen. Auch in diesem Fall wird der Zugriff über die Klasse `java.lang.Compiler` geregelt. Die dynamisch ladbare Bibliothek muß dann Routinen zum Zugriff auf den entsprechenden übersetzten Code besitzen.

4.3 Andere Hardware

Just in Time-Übersetzer, Natives und Fat Classfiles sind natürlich abhängig von der zugrundeliegen Hardware und Software. Aber sie verfolgen Konzepte, die systemunabhängig sind.

Eine andere Herangehensweise an das Problem Performancesteigerung ist der Austausch der Hardware selbst (vgl. Abbildung 4.4).

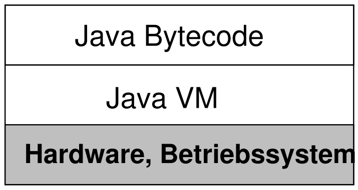

Abbildung 4.4: Abstrakte Ebenen von Java: Hardware, Betriebssystem

4.3.1 Teilauslagerung

Teile einer JVM lassen sich relativ einfach auf andere Hardwaresysteme oder Subsysteme ausgliedern. Dazu sind nur ähnlich der in Abschnitt 4.1 besprochen `native` Methoden die von der JVM verwendeten Routinen auf dem Hostsystem durch effizientere (z.B. einen Coprozessor benutzende) zu ersetzen.

Beispielsweise ließen sich die Klassen-Methoden für trigonometrische Funktionen in `java.lang.Math` so mit wenig Aufwand durch leistungsfähigere Äquivalente ersetzen.

4.3.2 Vollimplementierung

Aber auch die vollständige Implementierung der JVM ist denkbar. Die JVM stellt eine sehr kleine, robuste Menge von Befehlen auf einer sehr einfachen abstrakten Maschinenstruktur zur Verfügung

Die Konzepte (vgl. Kapitel 2) lassen sich auf bestehende Hardware abbilden. Und durch Optimierung und Ausnutzung besonderer Hardwarebedingungen läßt sich die Performance deutlich steigern.

Weiter Überlegungen zur Portierung der Runtime der JDK 1.0.2 findet man in [Günt97].

Kapitel 5

Portierung von Bibliotheken

Die meisten nativen Routinen sind im Bibliothekssystem der Java-API zu finden. Die Kernbibliotheken der JDK 1.0.x[1] sind `java.io`, `java.net` und `java.awt`, das Abstract Window Toolkit (AWT). Diese beinhaltet Funktionen für eine grafisches Benutzerinterface (GUI) und für grafische Ausgaben.

5.1 Abstract Window Toolkit

Das AWT ist eine portable GUI-Bibliothek für Java. Diese kann man in einem Java Programm durch den Befehl `import java.awt.*` einbinden.

Das AWT soll nachfolgend als Beispiel näher beschrieben werden, um Probleme und Lösungsmöglichkeiten für eine Portierung oder Implementierung bei schon laufendem Runtime zu zeigen.

5.1.1 java.awt

Das AWT, im package `java.awt,` bietet Funktionen für die Gestaltung von graphischen Benutzerinterfaces. Neben den reinen GUI-Komponenten enthält es aber auch grundlegende Klassen zur Grafikausgabe. Das OMT-Diagramm in Abbildung 5.1 zeigt einige der Klassen und ihre Beziehungen[2].

Als GUI-Klassen gibt es Komponenten und Layoutmanager. Komponenten sind entweder einzelne Objekte, wie `Button`, `Label`, `List`, etc., oder Container, die mehrerer Komponenten beinhalten. `LayoutManager` kontrollieren das Layout der Elemente innerhalb eines Containers.

Wichtigste Grafik-Klasse ist `java.awt.Graphics`, die ist als abstrakte Klasse definiert ist und eine systemunabhängige Schnittstelle zur Grafik herstellt. `Graphics`-Objekte können nicht direkt erzeugt werden, aber sie können über die `getGraphics()`-Routine einer Komponete angefordert werden.

[1] Mit Einführung der JDK 1.1 beta hat JavaSoft diese wesentlich erweitert.

[2] Für eine Gesamtübersicht aller Klassen ist der Referenzteil von [Flan96] zu empfehlen. Eine Einführung in das AWT bietet [Yu95].

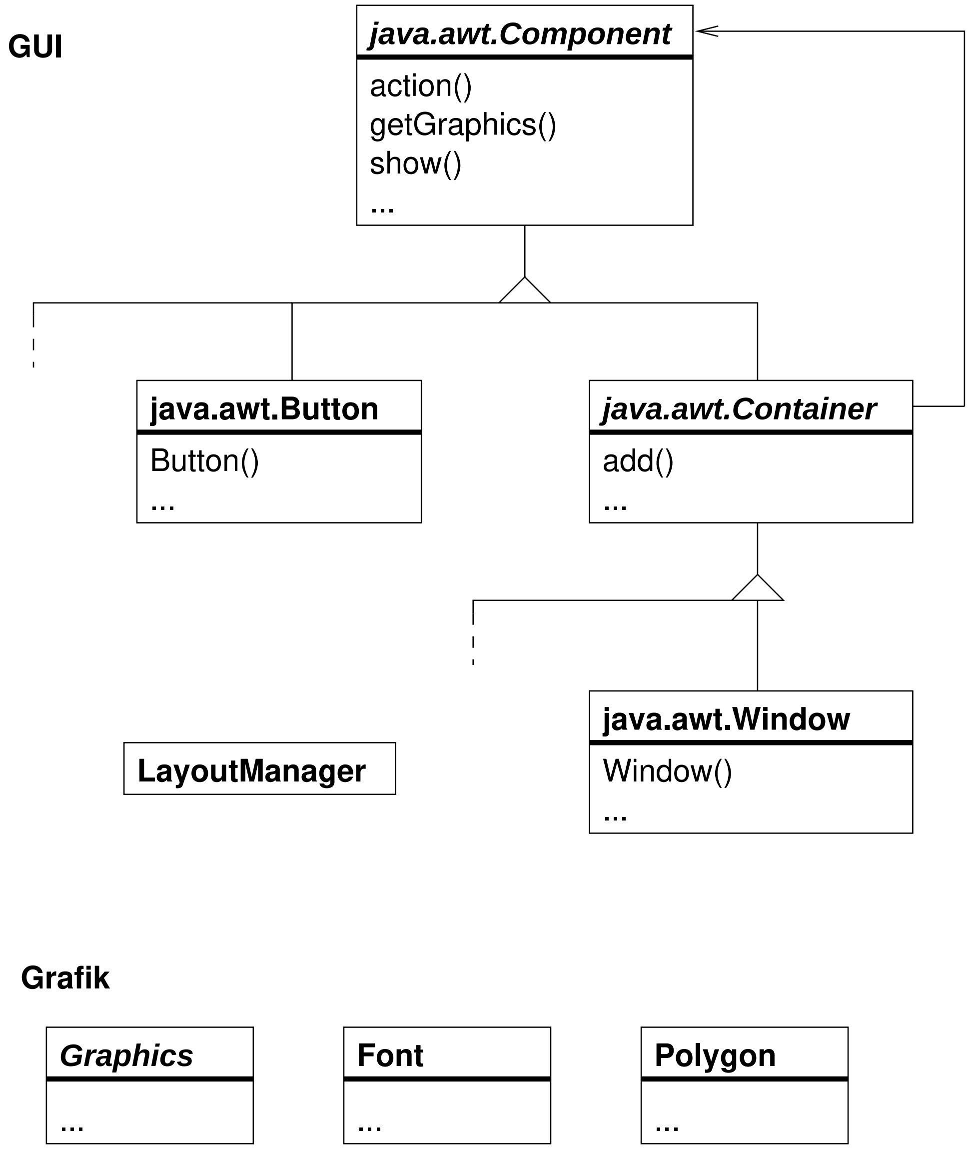

Abbildung 5.1: OMT-Diagramm: `java.awt.*`

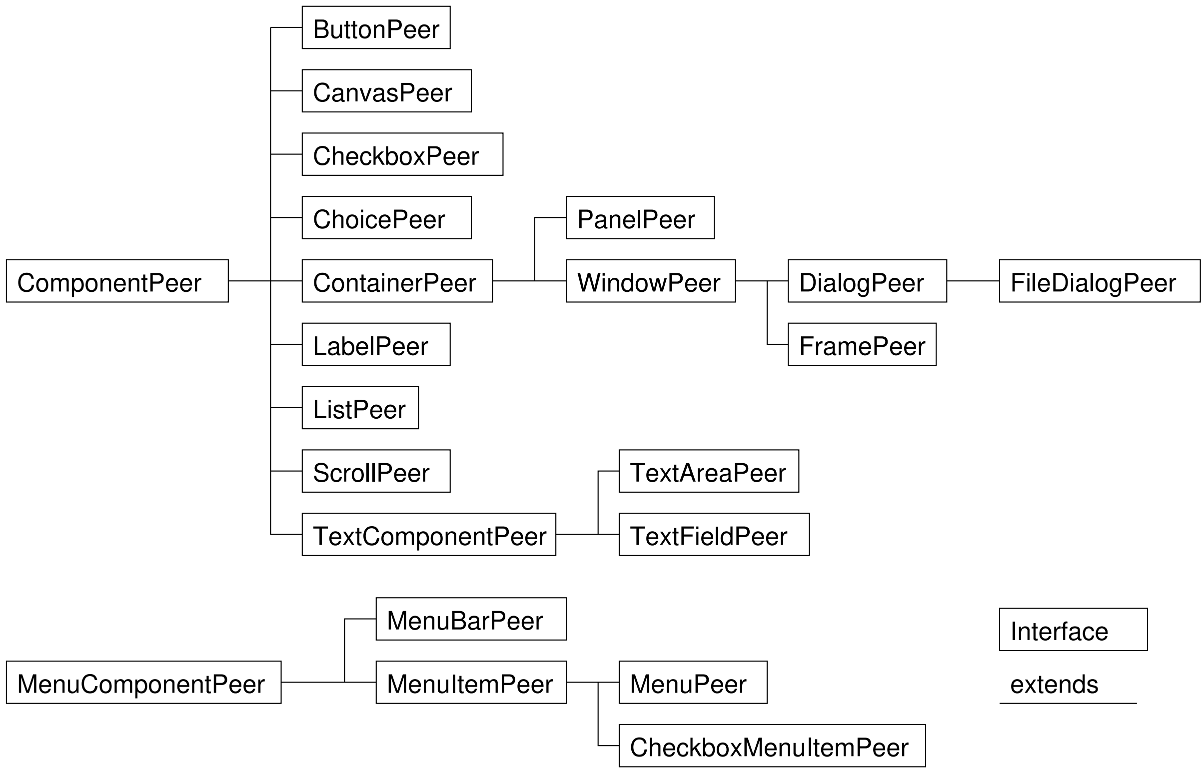

Abbildung 5.2: Struktur der `java.awt.peer`-Interfaces

5.1.2 java.awt.peer

Das Package `java.awt.peer` besteht vollständig aus Interface-Definitionen (vgl. Abbildung 5.2). Die Hirachie entspricht fast vollständig der des Packages `java.awt`. Ziel der Peer-Spezifikationen ist es, eine abstrakte Schnittstelle für die GUI-Implementierung auf verschiedenen Plattformen zu bieten (vgl. Abschnitt 2.3).

5.1.3 sun.awt.win32

Bei Implementierung des AWT gilt es, die in Abschnitt 5.1.1 beschriebene Klassenstruktur und Funktionalität zur Verfügung zu stellen.

Sun hat für die Einbindung des AWT eine abstrakte Klasse `java.awt.Toolkit` vorgesehen. Eine AWT-Implementation muß diese erweitern und die entsprechenden Funktionen besitzen. Bei Benutzung von AWT-Objekten wird dieses Toolkit *einmal* instanziert und mit seinen Funktionen das entsprechende Objekt zurückgeliefert. Im Diagramm in Abbildung 5.3 wird ein kleiner Ausschnitt der von Sun entwickelten Toolkits für Windows und Motif dargestellt.

Wird in einem Java-Programm ein Button-Objekt der Klasse `java.awt.Button` eingebunden, erzeugt das Windows-Toolkit ein Objekt der Klasse `sun.awt.win32.MButtonPeer` und gibt dies zurück. Damit die tatsächliche Implementierung der Grafik und GUI-

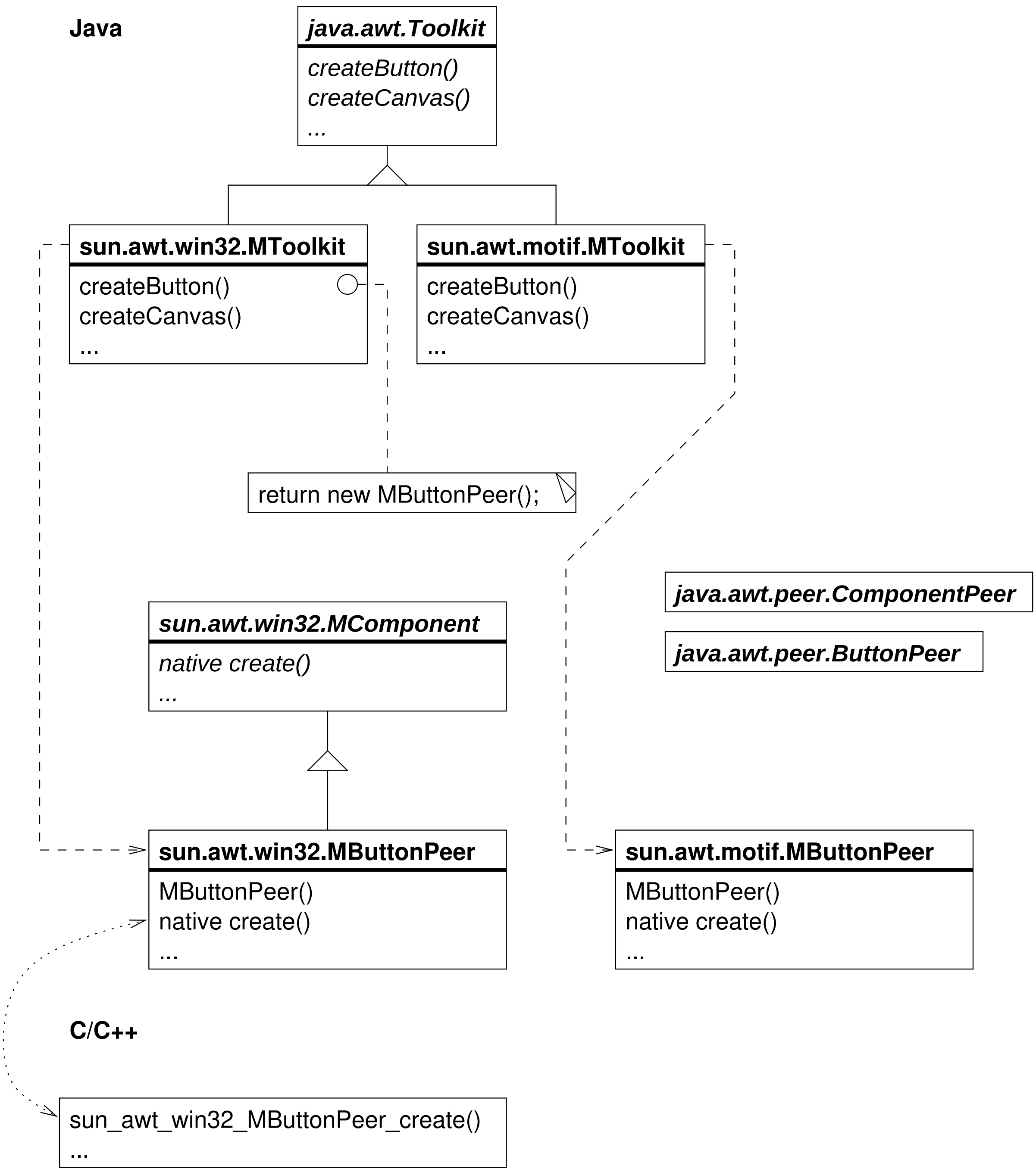

Abbildung 5.3: OMT-Diagramm: `sun.awt.*.MToolkit`

Objekte mit denen der AWT-Spezifikation übereinstimmt, müssen diese Klassen die Interfaces des `java.awt.peer` Paketes unterstützen. Die Klasse `sun.awt.win32.MButtonPeer` implementiert zum Beispiel das Interface `java.awt.peer.ButtonPeer`.

Ein großer Teil der AWT ist somit in Java selbst geschrieben, nur um tatsächliche Objekte auf dem Bildschirm zu erzeugen, werden ca. 140 native Methoden benötigt.

Die 1.0.x Implementierung von Sun baut unter Solaris auf das Motif-Framework[3] und unter Windows auf die Microsoft Foundation Classes (MFC) auf. Eine Klasse `MButtonPeer` mit ihrer nativen Methode `create()` wird deshalb auf das entsprechende C++-Äquivalent der MFC mit den entsprechenden Routinen abgebildet. Das nachfolgende Programmsegment zeigt den C++-Code.

```
/* C++-Klasse AwtButton erbt von MFC CButton */

class AwtButton :  public CButton
{
public:
        AwtButton();
        ~AwtButton();

        /* ...  */
}
```

Die im OMT-Diagramm (Abbildung 5.3) angedeutete C-Funktion `sun_awt_win32_MButtonPeer_create()` fungiert sozusagen als Schnittstelle zwischen dem in Java und in C++ geschriebenen Code.

Auf Grund dieser Struktur der `sun.awt.win32` hängt der Aufwand, die AWT-Bibliothek der JDK 1.0.x von Windows auf andere Systeme zu portieren, sehr von den dortigen Gegebenheiten ab. Mit einem Fenster-Framework, daß in Struktur, Umfang und Leistungsfähigkeit den MFC entspricht, wäre es möglich, den Sun-Code 1:1 umzusetzen. Der in Java geschriebene Teil könnte ohne Änderungen übernommen werden, und der C++-Code muß nur an die neuen GUI-Objekte angepasst werden. Da die AWT nur eine relativ kleine Standard-GUI-Menge darstellt, ist dies oft möglich.

Wenn ein entsprechendes Framework nicht existiert, oder es eine andere Struktur besitzt, liegt der Hauptaufwand der Portierung in der Abbildung des AWT.

Mit der Freigabe der 1.1-Beta-Release der JDK hat JavaSoft den AWT-Windows-Teil vollkommen neu geschrieben[4] und unabhängig von den Microsoft Foundation Classes gestaltet. Derzeit ist diese Software aber noch im öffentlichen Beta-Test und noch nicht fehlerfrei[5].

[3]Motif ist eine Bibliothek zur Darstellung eines grafischen Benutzerinterfaces unter X.

[4]http://www.javasoft.com/products/jdk/1.1/

[5]http://www.javasoft.com/products/jdk/1.1/README

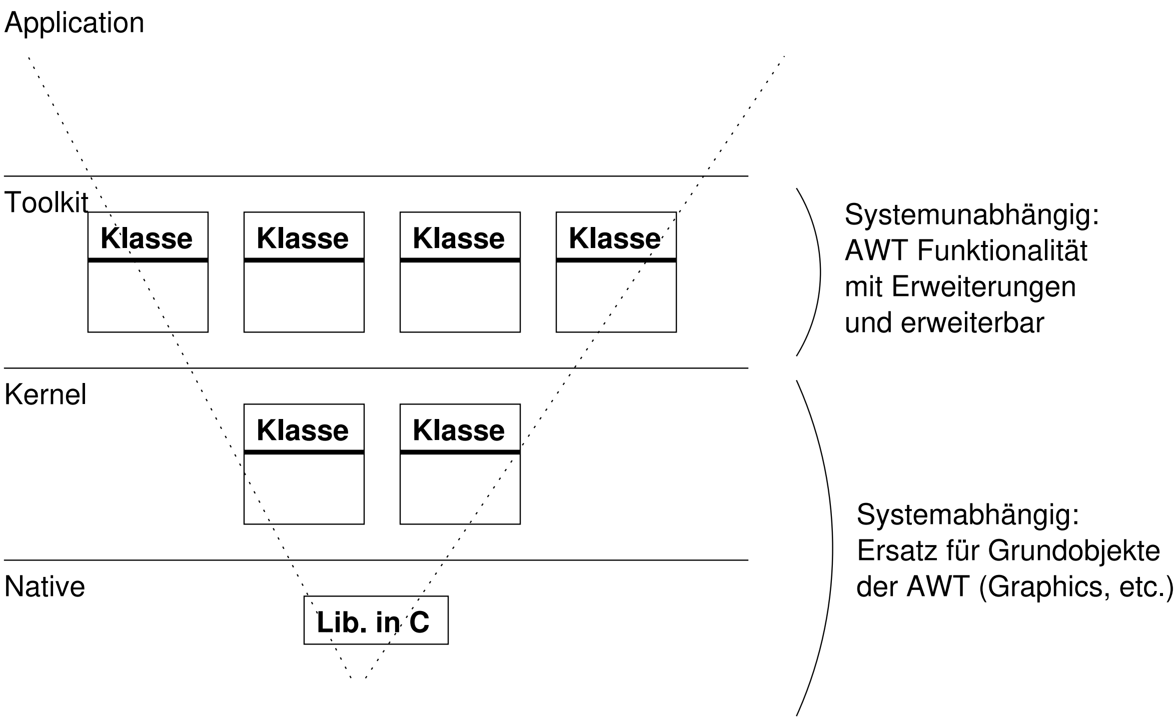

Abbildung 5.4: Baumstruktur der BISS-AWT

5.1.4 Alternativen

BISS-AWT

Auf der Mailingliste für die unter der GNU General Public License[6] erhältliche JVM kaffe hat Anfang 1997 eine für Unix-Plattformen mit Source-Code frei erhältliche AWT-Implementierung das Interesse auf sich gezogen, das BISS AWT[7].

Ursprünglich nur als Erweiterung und Anpassung des AWT an eigene Zwecke gedacht, hat der Hersteller (BISS GmbH) das BISS Java Framework entworfen, das aufbauend auf die Basisklassen `Graphics`, `Frame`, `Panel` und `Canvas` in `java.awt` eine selbständige graphische Benutzerschnittstelle definiert [BISS97].

In Zusammenhang mit der Änderung der Lizenzbedingungen bei Freigabe der JDK 1.0.2 und den Überlegungen der Internetgemeinde zu einer frei erhältlichen JVM[8], haben die Entwickler aufbauend auf das Framework die `java.awt.peer` Interfaces, sowie die für ein Stand-alone- System nötigen Grundklassen implementiert. Als Referenzsystem wurde Linux/X mit kaffe und den Xlib-Funktionen gewählt.

Ein Vorteil, den diese AWT-Implementierung gegenüber dem Sun-Original besitzt ist die große Systemunabhängigkeit (vgl. Abbildung 5.4): nur 1/3 der für die Sun AWT zu implementierenden Native-Methoden sind nötig. Durch einen konsequent hirachischen Aufbau

[6] http://www.gnu.org/copyleft/gpl.html
[7] http://www.biss-net.com/biss-awt.html
[8] JOLT, Java Open Language Toolkit: http://www.redhat.com/jolt/

werden alle GUI-Komponenten in Java implementiert und eben nur wenige Basisklassen mit systemabhängigen Aufrufen gebraucht.

SAWT

Ein anderer Ansatz ein freies Abtract Window Toolkit für Unix/X zu entwickeln ist das SAWT-Projekt[9]. Hierbei werden ebenfalls alle Peer-Interfaces implementiert. Aber anders als das Sun-Entwicklerteam werden nicht das Motif-Framework, sondern direkt Xlib-Aufrufe verwendet.

Interessant an diesem Projekt ist die Herangehensweise; in vier Schritten soll die volle Funktionalität zur Verfügung gestellt werden.

A: API Gerippe

B: minimale Funktionalität (Anzeige)

C: ganze Funktionalität (Fonts, Farben, etc.)

D: optimierte Funktionen

SAWT befindet sich immer noch im ALPHA-Zustand, bei fast allen Komponeten ist erst Entwicklungsstand B (minimale Funktionalität) erreicht, und ist deshalb noch keine Alternative zur Sun-Implementation.

5.2 Weiterführendes

Das Abtract Window Toolkit stellt derzeit die größte zu portierende Bibliothek der JDK dar. Der Aufwand hierfür hängt stark von den Gegebenheiten auf der Zielplattform ab. Weiter Bibliotheken sind `java.net` oder `sun.audio` die in der JDK 1.0.x größtenteils in Java implementiert sind und deshalb weniger Portierungsaufwand erfordern.

Neben Java-Compiler und Interpreter sind die Kernbibliotheken Teil des Java-Systems (vgl. dazu Abschnitt 2). Sie stehen auf jeder unterstützten Plattform zur Verfügung, und JavaSoft allein ist für die Definition der Schnittstellen zuständig.

Mit Einführung der JDK 1.1 hat JavaSoft diese unter anderem um die *JDBC*[10] und die *Java Security API*[11] erweitert. JDBC ist die Datenbankschnittstelle für Java, die Java-Programme den Zugriff auf rationale Datenbanken erlaubt. Die neue Java Security API enthält Funktionen für digitale Unterschriften, Verschlüsselung und Authentisierung.

Mit Implementierung dieser Kernbibliotheken besitzt Java eine auf allen Plattformen gleiche Umgebung und ermöglicht eine Softwareentwicklung für ein einheitliches System.

[9] http://slhp1.epfl.ch/sawt.html
[10] http://www.javasoft.com/products/jdk/1.1/docs/guide/jdbc/index.html
[11] http://www.javasoft.com/products/jdk/1.1/docs/guide/JavaSecurityOverview.html

Kapitel 6

Zusammenfassung und Ausblick

Die Programmiersprache Java ist innerhalb sehr kurzer Zeit zu einer festen Größe in der Softwareentwicklung geworden, und besonders leistungsfähige JIT-Übersetzer — auf möglichst vielen Rechnerplattformen — werden dies weiter unterstützen.

Die Benchmarks dieser Studienarbeit bestätigen eine durchschnittliche Leistungssteigerung von einem Faktor 5. Einziger Schwachpunkt aller JITC-Implementierungen ist die Erzeugung von Objekten. Im Vergleich mit dem Interpreter der JDK ergeben fast alle JVM mit JIT-Übersetzung hier eine langsamere Ausführungszeit.

Eine vollständige Implementierung der Java Virtual Machine besteht neben Java-Compiler und Bytecode-Interpreter auch aus den Kernbibliotheken des Java-Systems. Ein Teil ist das Abstract Window Toolkit (AWT).

Der Aufwand zur Portierung des AWT der JDK 1.0.2 hängt, durch die Struktur der Sun-Implementierung für Windows bedingt, stark von den Voraussetzungen auf der Zielplattform ab. Nur bei ähnlichen Grundbedingungen kann viel Code 1:1 übernommen werden.

Zusammen mit der Implementierung weiterer API-Schnittstellen vervollständigt dies das Java-System, und ergibt eine auf allen Plattformen standardisierten Laufzeit- und Programmierumgebung.

Anhang A

Programme

Benchmark

```
/** SMall JVM-Benchmark-Collection
  * @version 1.1
  * @author Marc Schanne <Marc.Schanne@stud.uni-karlsruhe.de>
  */

import java.applet.*;
import java.awt.*;
import java.io.*;

/** Benchmark ist eine abstrakte Klasse, die fuer konkrete
  * Benchmark-Programme Methoden zur Zeitmessung, systemunabh.
  * Ausgabe und Appletdarstellung zur Verfuegung stellt.
  * runBenchmark() und infoBenchmark() muessen vom konkreten
  * Testprogramm implementiert werden.
  */

public abstract class Benchmark extends Applet {

  // Variablen zur Zeitmessung und Ausgabe

  private long startTime, elapsedTime;
  private Button runButton;
  private TextArea textArea;

  // Applet oder Application?

  private boolean isApplet = false;
```

```
/** System- (d.h.  von Applet oder Application)
  * unabhaengige Ausgabe-Routine
  */

public void printBenchmark(String str) {
  if (isApplet)
    textArea.appendText(str + "\n");
  else
    System.out.println(str);
}

/** Zeitklammer oeffnen (d.h Startzeit speichern) */

public void beginBenchmark() {
  System.gc();
  printBenchmark("Die Uhr laeuft ...");
  startTime = System.currentTimeMillis();
}

/** Zeitklammer schliessen und Laufzeit ausgeben */

public void endBenchmark() {
  elapsedTime = System.currentTimeMillis() - startTime;
  double seconds = elapsedTime / 1000.0;
  printBenchmark{"...  und stop.  Laufzeit:  " + seconds
                 + " Sek.");
}

/** Der Test selbst;
  * Ausgaben ueber die printBenchmark()-Methode
  * Zeitmessung mit begin/endBenchmark()
  */

public abstract void runBenchmark();

/** Gibt Informationen ueber den Test aus */

public abstract void infoBenchmark();

/*  Methoden fuer Appletdarstellung */

/** Einrichten des Start-Buttons und des Ausgabe-TextAreas
  * und Informationen ueber den Test ausgeben.
  */

public void init() {
  isApplet = true;                    // Ausgabe in Applet
  this.setLayout(new BorderLayout(0, 0));
  runButton = new Button("Benchmark starten");
  this.add("North", runButton);
```

```
    textArea = new TextArea(15, 60);
    textArea.setFont(new Font("Helvetica", Font.PLAIN, 12));
    textArea.setEditable(false);
    this.add("Center", textArea);

    this.show();                          // alles anzeigen
    infoBenchmark();
  }

  /** Bei Klick auf den Start-Button wird der Benchmark
    * ausfuehren, andere Events werden in der Event-Kette
    * weitergeben.
    * Es gibt keine Abfrage ob ein Test bereits laeuft!
    */

  public boolean action(Event e, Object o) {
    if (e.target == runButton) {
      runBenchmark();
      return true;
    }
    return super.action(e, o);
  }
}
```

MatrixBenchmark

```
/** SMall JVM-Benchmark-Collection:  Matrizenrechnung
  * @version 1.1
  * @author Marc Schanne <Marc.Schanne@stud.uni-karlsruhe.de>
  */

import java.util.*;

/** Matrizen(Zeilen x Spalten)-Klasse mit
  * Multiplikation und Gauss-Elimination
  */

class Matrix {
  public double elem[][];              // Matrix-Elemente
  public int rows;                     // Anzahl der Zeilen und
  public int cols;                     // Spalten

  /** Erzeugen der Matrix(rows x cols) */

  public Matrix(int r, int c) {
    rows = r;
    cols = c;
    elem = new double[rows][cols];
  }

  /** Konstruieren einer erweiterten Matrix fuer die Gauss-
    * Elimination aus einer quadradtischen Matrix A und einem
    * Vektor b.
    */

  public Matrix(QMatrix a, VMatrix b) {
    rows = a.rows;
    cols = a.cols+1;
    elem = new double[a.rows][a.cols+1];
    int i, j;

    /* Verwendung der schnellen System-Routine arraycopy().
     * Vgl.  dazu Abschnitt 3.3
     */

    try {
      for (i = 0; i < rows; i++)
        System.arraycopy(a.elem[i], 0, elem[i], 0, a.cols);
    }
    catch(ArrayIndexOutOfBoundsException e) { }
    catch(ArrayStoreException e) { }
```

```
  for (i = 0; i < rows; i++)
    elem[i][a.cols] = b.elem[i][0];
}

/** Matrix mit zufaelligen Double-Werten fuellen */

public final void randomFill() {
  Random random = new Random();
  int i, j;
  for (i = 0; i < rows; i++)
    for (j = 0; j < cols; j++)
      elem[i][j] = random.nextDouble();
}

/** Einfache Matrizenmultiplikation in O(n^3) */

public final Matrix mult(Matrix b) {
  int i, j, k;
  Matrix c = new Matrix(rows, b.cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < b.cols; j++) {
      for (k = 0; k < cols; k++) {
        c.elem[i][j] += elem[i][k] * b.elem[k][j];
      }
    }
  }
  return c;
}

/** Vorwaerts-Elimination nach Gauss mit Pivot-Suche */

public final void eleminate() {
  double[] elem_i, elem_max, elem_j;  // Vgl.  Abschnitt 3.3
  double dummy;
  int i, j, k, max;
  for (i = 0; i < rows; i++) {
    max = i;
    for (j = i+1; j < rows; j++)
      if (Math.abs(elem[j][i]) >
          Math.abs(elem[max][i])) max = j;
    elem_i = elem[i];
    elem_max = elem[max];
    for (k = i; k < cols; k++) {
      dummy = elem_i[k];
      elem_i[k] = elem_max[k];
      elem_max[k] = dummy;
    }
```

```
      for (j = i+1; j < rows; j++) {
        elem_j = elem[j];
        for (k = cols-1; k >= i; k--)
          elem_j[k] -= elem_i[k] * elem_j[i] / elem_i[i];
      }
    }
  }

  /** Rueckwaerts-Einsetzen (Gauss)
    * Der Loesungsvektor wird erzeugt und zureckgegeben.
    */

  public final VMatrix substitute() {
    VMatrix x = new VMatrix(rows);
    double dummy;
    int i, k;
    for (i = rows-1; i >= 0; i--) {
      dummy = 0.0;
      for (k = i+1; k < cols-1; k++)
        dummy += elem[i][k] * x.elem[k][0];
      x.elem[i][0] = (elem[i][cols-1]-dummy) / elem[i][i];
    }
    return x;
  }

  /** Ueberschreibt Object.toString */

  public String toString() {
    int i, j;
    StringBuffer dummy = new StringBuffer("");
    for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
        dummy.append(elem[i][j] + ", ");
      dummy.append("\n");
    }
    String out = new String(dummy);
    return out;
  }
}

/** Klasse quadratischer Matrizen
  * (direkte Kindklasse normalerMatrizen)
  * Für quad.  Matrizen kann der Gauss-Alg.  aufgerufen werden.
  */

class QMatrix extends Matrix {
```

```
  public QMatrix(int r) {
    super(r, r);
  }

  /** Gauss, verwendet eleminate(), substitute() */

  public final VMatrix gauss(VMatrix y) {
    Matrix ay = new Matrix(this, y);
    ay.eleminate();
    VMatrix x = ay.substitute();
    return x;
  }
}

/** Vektoren sind ebenfalls als Kindklasse von Matrix
  * implemntiert.
  */

class VMatrix extends Matrix {
  public VMatrix(int r) {
    super(r, 1);
  }
}

/** Benchmark mit grossen Matrizen
  * Multiplikation und Gauss (keine optimierten Alg.)
  */

public class MatrixBenchmark extends Benchmark {

  public void runBenchmark() {
    printBenchmark("Matrixmutliplikation in O(n^3)");
    QMatrix a = new QMatrix(200);
    QMatrix b = new QMatrix(200);
    a.randomFill(); b.randomFill();
    Matrix c;
    beginBenchmark();                           // Test-Start
    c = a.mult(b);                              // C = A * B
    endBenchmark();                             // Test-Ende
    printBenchmark("Gauss-Elimination fuer LGS");
    QMatrix m = new QMatrix(400);
    VMatrix y = new VMatrix(400);
    VMatrix x;
    m.randomFill(); y.randomFill();
    beginBenchmark();                           // Test-Start
    x = m.gauss(y);                             // A * x = y
    endBenchmark();                             // Test-Ende
  }
```

```
  public void infoBenchmark() {
    printBenchmark("SMall JVM-Benchmark-Collection:  "
                  + Matrizenrechnung");
    printBenchmark("v1.1");
    printBenchmark("Marc Schanne "
                  + "<Marc.Schanne@stud.uni-karlsruhe.de>\n");
  }

 /** Application-Main
   * Benchmark-Objekt erzeugen und Test ausfuehren
   */

  public static void main(String args[]) {
    MatrixBenchmark mb = new MatrixBenchmark();
    mb.infoBenchmark();
    mb.runBenchmark();
  }
}
```

PolyBenchmark

```
/** SMall JVM-Benchmark-Collection:  Polynome
  * @version 1.1
  * @author Marc Schanne <Marc.Schanne@stud.uni-karlsruhe.de>
  */

import java.util.*;


/** Polynom n-ter Ordung mit Multiplikation */

class Poly {
  private double elem[];           // Koeffizentenfeld

  /** double-Feld für Koeffizenten mit Laenge n erzeugen */

  public Poly(int n) {
    elem = new double[n];
  }

  /** Polynom der Laenge n erzeugen und die ersten Elemete aus
    * Polynom a kopieren.
    */

  private Poly(Poly a, int n) {
    int i;
    elem = new double[n];

    // Vgl.  Abschnitt 3.3
    try {
      System.arraycopy(a.elem, 0, elem, 0, a.elem.length);
    }
    catch(ArrayIndexOutOfBoundsException e) { }
    catch(ArrayStoreException e) { }

    for (i = a.elem.length; i < n; i++)
      elem[i] = 0;
  }

  /** Zufaelliges Fuellen mit Werten */

  public final void randomFill() {
    int i, len;
    Random random = new Random();
    len = elem.length;
    for (i = 0; i < len; i++)
      elem[i] = random.nextDouble();
  }
```

```
/** Rekursive Funktion zum Multipizieren der Polynome this
  * und b.
  * Beschr.:  Nach dem Prinzip "Teile und Herrsche" wird
  *           das Problem der Multiplikation von Polynomen der
  *           Groesse n durch Loesen von 3 Teilproblemen der
  *           Groesse n/2 geloest.
  *           Fuer n=1 ist das Produkt gleich dem Skalar-
  *           produkt der beiden Konstanten-Koeffizienten.
  * Aufwand:  O(n^lg3)
  * Randbed:  this.elem.length == b.elem.length == 2^k
  *           (wird deshalb aus mult() aufgerufen)
  */

private final Poly multRec(Poly b) {
  if (elem.length == 1) {
    Poly c = new Poly(1);
    c.elem[0] = elem[0] * b.elem[0];
    return c;
  }
  else {
    int len;
    int len2 = elem.length / 2;       // Zerlegung in 6 Polynome
    Poly al = new Poly(len2);         // der Laenge n/2:
    Poly bl = new Poly(len2);         // al, bl, ah, bh, t1, t2

    try {
      System.arraycopy(elem, 0, al.elem, 0, len2);
      System.arraycopy(b.elem, 0, bl.elem, 0, len2);
    }
    catch(ArrayIndexOutOfBoundsException e) { }
    catch(ArrayStoreException e) { }

    Poly ah = new Poly(len2);
    Poly bh = new Poly(len2);

    try {
      System.arraycopy(elem, len2, ah.elem, 0, len2);
      System.arraycopy(b.elem, len2, bh.elem, 0, len2);
    }
    catch(ArrayIndexOutOfBoundsException e) { }
    catch(ArrayStoreException e) { }

    Poly t1 = new Poly(len2);
    Poly t2 = new Poly(len2);
    for (int i = 0; i < len2; i++) {
      t1.elem[i] = al.elem[i] + ah.elem[i];
      t2.elem[i] = bl.elem[i] + bh.elem[i];
    }

    Poly rm = new Poly(elem.length);    // Teilberechnungen
    rm = t1.multRec(t2);
```

```
    Poly rl = new Poly(elem.length);
    rl = al.multRec(bl);
    Poly rh = new Poly(elem.length);
    rh = ah.multRec(bh);

    Poly c = new Poly(2*elem.length-1); // Rekombination

    len = elem.length-1;

    try {
      System.arraycopy(rl.elem, 0, c.elem, 0, len);
      System.arraycopy(rh.elem, 0, c.elem, len+1, len);
    }
    catch(ArrayIndexOutOfBoundsException e) { }
    catch(ArrayStoreException e) { }

    c.elem[len] = 0;

    for (int i = 0; i < len; i++)
      c.elem[len2+i] += rm.elem[i] - (rl.elem[i] + rh.elem[i]);
    return c;
  }
}

/** Mult hat 2 Aufgaben:
  * 1) Aufruf des Multiplikations-Alg.  (mutlRec()) mit
  *    Polynomen richtiger Groesse:  n = 2^(aufrunden(log_2(n))
  * 2) Rueckgabe des Ergebnisses
  */

public final Poly mult(Poly b) {
  Poly a2 = new Poly(this, (int)Math.pow(2, (double)((int)(
     Math.log (Math.max(elem.length, b.elem.length)) /
     Math.log(2.0)) + 1)));
  Poly b2 = new Poly(b, (int)Math.pow(2, (double)((int)(
     Math.log(Math.max(elem.length, b.elem.length)) /
     Math.log(2.0)) + 1)));
  return a2.multRec(b2);
}

/** Ueberschreibt Object.toString() */

public String toString() {
  StringBuffer dummy = new StringBuffer("p(x) = ");
  for (int i = elem.length-1; i > 0; i--)
    if (elem[i] != 0.0)
      dummy.append(elem[i] + "*x^" + i + " + ");
```

```
    dummy.append(elem[0]);
    String out = new String(dummy);
    return out;
  }
}

/** Benchmark:  Polynom-Multiplikation */

public class PolyBenchmark extends Benchmark {

  /** der Test */

  public void runBenchmark() {
    printBenchmark("c(x) = a(x) * b(x); a, b vom Grad 4000");
    Poly a = new Poly(4000);
    Poly b = new Poly(4000);
    Poly c;
    a.randomFill(); b.randomFill();
    beginBenchmark();                          // Test-Start
    c = a.mult(b);                             // c(x) = a(x) * b(x)
    endBenchmark();                            // Test-Ende
  }

  /** Versionsnummer und Name */

  public void infoBenchmark() {
    printBenchmark("SMall JVM-Benchmark-Collection:  "
                  + "Polynom-Multiplikation");
    printBenchmark("v1.1");
    printBenchmark("Marc Schanne "
                  + "<Marc.Schanne@stud.uni-karlsruhe.de>\n");
  }

  /** Main-Prog.  fuer Application
    * Erzeugen des Benchmark-Objekts und Ausführen des Tests
    */

  public static void main(String argv[]) {
    PolyBenchmark pb = new PolyBenchmark();
    pb.infoBenchmark();
    pb.runBenchmark();
  }
}
```

PrimeBenchmark

```
/** SMall JVM-Benchmark-Collection:  Primzahlen
  * @version 1.1
  * @author Marc Schanne <Marc.Schanne@stud.uni-karlsruhe.de>
  */

import java.util.*;

/** PrimeList stellt eine Prime-Flag-Liste fuer die Zahlen
  * von 2 bis len und eine Methode zur Primzahlenberechnung
  * mit dem Sieb des Eratosthenes zur Verfuegung.
  */

class PrimeList {
  private boolean flag[];           // Prime-Flag fuer 2 ...
  private int sqrtLen;              // ...  bis sqrtLen^2

  /** @param sqrtLen Quadrat-Wurzel der gewuenschten
    *        Array-Laenge
    */

  public PrimeList(int sqrtLen) {
     this.sqrtLen = sqrtLen;
     flag = new boolean[sqrtLen*sqrtLen+1];
     int i, len;
     len = flag.length;             // Annahme:  alle prim
     for(i = 0; i < len; i++) flag[i] = true;
  }

  /** Primzahlenberechnug mit Sieb der Eratosthenes
    * von 2 bis sqrtLen^2
    *
    * Bsp.  fuer sqrtLen=4:
    * Init. 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    * i=2   2 3 - 5 - 7 - 9  - 11  - 13  - 15  -
    * i=3   2 3 - 5 - 7 - -  - 11  - 13  -  -  -
    * i=4 keine Primzahl.  Fertig.
    */

  public final void runSieve() {
     int i, k, prime;
     for (i = 2; i <= sqrtLen; i++) {
       if(flag[i]) {
         prime = i;
```

```
        for (k = i + prime; k < flag.length; k += prime)
          flag[k]=false;
      }
    }
  }

  /** zur einfachen Ausgabe, ueberschreibt Object.toString() */

  public String toString() {
    int i, len;
    StringBuffer dummy = new StringBuffer("Primz.  von 2 bis ");
    dummy.append((flag.length-1) + ":  ");
    len = flag.length;
    for (i = 2; i < len; i++)
      if (flag[i])
        dummy.append(i + " ");
    String out = new String(dummy);
    return out;
  }
}

/** Benchmark mit Primzahlenwaesche des Eratosthenes */

public class PrimeBenchmark extends Benchmark {

  /** Erzeugt ein PrimeList-Objekt und ruft das Primzahlensieb
    * dafuer auf.
    */

  public void runBenchmark() {
    printBenchmark("1) Erzeugen des PrimeList-Objects (im "
                   + "wesentlichen ein boolean-Array fuer die "
                   + "Zahlen 2 bis 9.000.000)");
    beginBenchmark();                          // Test-Start
    PrimeList pl = new PrimeList(3000);        // Objekt erzeugen
    endBenchmark();                            //Test-Ende
    printBenchmark("2) Primzahlenberechnung selbst");
    beginBenchmark();                          // Test-Start
    pl.runSieve();                             // Primzahlensieb
    endBenchmark();                            // Tes-Ende
  }

  public void infoBenchmark() {
    printBenchmark("SMall JVM-Benchmark-Collection:  "
                   + "Primezahlensieb");
```

```
    printBenchmark("v1.1");
    printBenchmark("Marc Schanne"
                   + "<Marc.Schanne@stud.uni-karlsruhe.de>\n");
  };

  /** Application-Main */

  public static void main(String argv[]) {
    PrimeBenchmark pb = new PrimeBenchmark();
    pb.infoBenchmark();
    pb.runBenchmark();
  }
}
```

SortBenchmark

```
/** SMall JVM-Benchmark-Collection:  Integerfeld sortieren
  * @version 1.1
  * @author Marc Schanne <Marc.Schanne@stud.uni-karlsruhe.de>
  */

import java.util.*;

/** Liste von Integern mit verschiedenen elementaren
  * Sortier-Algorithmen (Bubble-Sort, Insertion-Sort und
  * Shell-Sort)
  */

class SortList {
  private int elem[];

  /** int-Feld der Laenge len erzeugen */

  public SortList(int len) {
    elem = new int[len];
  }

  /** Liste mit Zufallsintegerzahlen fuellen */

  public final void randomFill() {
    int i, len;
    Random random = new Random();
    len = elem.length;
    for (i = 0; i < len; i++)
      elem[i] = random.nextInt();
  }

  /** Bubble-Sort (Sortieren durch Austauschen)
    * Beschr.:  Durchlaufe immer wieder das feld und vertausche
    *           jedesmal, wenn es notwendig ist, benachbarte
    *           Elemente (wenn bei einem Durchlauf kein
    *           Austausch mehr erforderlich ist, ist die Datei
    *           sortiert).
    * Aufwand:  quadratisch
    */

  public final void sortBubbleSort() {
    int i, j, dummy;
    for (i = elem.length; i >= 0; i--)
      for (j = 1; j <= i; j++)
```

```
      if (elem[j-1] > elem[j]) {
        dummy = elem[j-1];
        elem[j-1] = elem[j];
        elem[j] = dummy;
      }
}

/** Insertion-Sort (Sortieren durch Einfuegen)
  * Beschr.:  Betrachte die Elemnte eines nach dem anderen und
  *           fuege jedes an seinen richtigen Platz zwischen
  *           den bereits betrachteten ein.  Das gerade
  *           betrachtete Element wird eingefuegt, indem die
  *           groessten Elemente einfach um eine Position nach
  *           rechts bewegt werden.
  * Aufwand:  quadratisch, aber fuer "fast sortierte" Felder
  *           linear
  */

public final void sortInsertionSort() {
  int i, j, dummy, len;
  len = elem.length;
  for (i = 1; i < len; i++) {
    dummy = elem[i];
    j = i;
    while ((j > 0) && (elem[j-1] > dummy)) {
      elem[j] = elem[j-1];
      j = j - 1;
    }
    elem[j] = dummy;
  }
}

/** Shell-Sort eine Verbesserung von Insertion-Sort
  * Beschr.:  Die Zahlenliste wird in mehreren Durchgaengen
  *           mit Insertion-Sort (vor-)sortiert.  Dabei
  *           werden Teilfolgen (mit bestimmtem Offset)
  *           sortiert.
  */

public final void sortShellSort() {
  int h = 1;                     // h-sortiert mit Folge
  int i, j, dummy, len;
  len = elem.length;
  do {
    h = 3 * h + 1;               // ..,1093,364,121,40,13,4,1
  } while (h < len);
```

```
    do {                           // Beginn Sortieralg.
      h = h / 3;
      for (i = h+1; i <= len; i++) {
        dummy = elem[i-1];
        j = i;
        while (elem[j-h-1] > dummy) {
          elem[j-1] = elem[j-h-1];
          j = j - h;
          if (j <= h) break;
        }
        elem[j-1] = dummy;
      }
    } while (h != 1);
  }

  /** Ueberschreibt Object.toString() */

  public String toString() {
    int i, len;
    StringBuffer dummy = new StringBuffer(" ");
    len = elem.length;
    for (i = 0; i < len; i++)
      dummy.append(elem[i] + " ");
    String out = new String(dummy);
    return out;
  }
}

/** Haeufiges Sortieren eines Integer-Feldes mit Shell-Sort */

public class SortBenchmark extends Benchmark {

  public void runBenchmark() {
    printBenchmark("1000 mal Array der Groesse 1000 in einer "
                 + "Methode mit Zufallszahlen fuellen und mit "
                 + "einer 2.  Methode sortieren");
    SortList sl = new SortList(1000);
    beginBenchmark();              // Test-Start
    for (int i=0; i < 1000; i++) { // Schleife !!!
      sl.randomFill();             // Fuellen mit Zufallszahlen
      sl.sortShellSort();          // Sortieren
    }
    endBenchmark();                // Test-Ende
  }
```

```
  public void infoBenchmark() {
    printBenchmark("SMall JVM-Benchmark-Collection:  Sortieren "
                   + "mit Shell-Sort");
    printBenchmark("v1.1");
    printBenchmark("Marc Schanne "
                   + "<Marc.Schanne@stud.uni-karlsruhe.de>\n");
  }

  /** Main-Prog.  fuer Application */

  public static void main(String args[]) {
    SortBenchmark sb = new SortBenchmark();
    sb.infoBenchmark();
    sb.runBenchmark();
  }
}
```

InitBenchmark

```
/** SMall JVM-Benchmark-Collection:  Objekte erzeugen
  * @version 0.1
  * @author Marc Schanne <Marc.Schanne@stud.uni-karlsruhe.de>
  */

import java.util.*;

/** Klasse mit Array-Objekt und einer Fuellmethode */

class Init {
  private double elem[];

  /** double-Feld mit Laenge n erzeugen */

  public Init(int n) {
    elem = new double[n];
  }

  /** Zufaelliges Fuellen mit Werten */

  public final void randomFill() {
    int i, len;
    Random random = new Random();
    len = elem.length;
    for (i = 0; i < len; i++)
      elem[i] = random.nextDouble();
  }
}

/** Benchmark:  Erzeugen von Objekten */

public class InitBenchmark extends Benchmark {

  public void runBenchmark() {
    int i, j;
    Init a[] = new Init[1000];
    printBenchmark(Ïnit as init can");
    beginBenchmark();                    // Test-Start
    for (j = 0; j < 100; j++)            // 100 x
      for (i = 0; i < 1000; i++)         // 1000 Objekte
        a[i] = new Init(1000);           // (= Arrays) erzeugen
    endBenchmark();                      // Test-Ende
  }
```

```
  public void infoBenchmark() {
    printBenchmark("SMall JVM-Benchmark-Collection:  "
                   + "Objekte erzeugen");
    printBenchmark("v0.1");
    printBenchmark("Marc Schanne "
                   + "<Marc.Schanne@stud.uni-karlsruhe.de>\n");
  }

  /** Main-Prog.  fuer Application */

  public static void main(String argv[]) {
    InitBenchmark ib = new InitBenchmark();
    ib.infoBenchmark();
    ib.runBenchmark();
  }
}
```

RecBenchmark

```
/** SMall JVM-Benchmark-Collection:  rekursiver Abstieg
  * @version 0.1
  * @author Marc Schanne <Marc.Schanne@stud.uni-karlsruhe.de>
  */

import java.util.*;


/** In einem grossen Array ermoeglicht diese Klasse einen
  * rekursive Abstieg
  */

class Rec {
  private double elem[];

  /** double-Feld mit Laenge n erzeugen */

  public Rec(int n) {
    elem = new double[n];
  }

  /** Zufaelliges Fuellen mit Werten */

  public final void randomFill() {
    int i, len;
    Random random = new Random();
    len = elem.length;
    for (i = 0; i < len; i++)
      elem[i] = random.nextDouble();
  }

  /* rekursiver Aufruf von doRec */

  public final double doRec(int l, int r) {
    if (l == r) {                     // Abbruchbed.
      return elem[l];
    }
    else {
      return doRec(l, l+(int)((r-l)/2)) *
                      doRec(l+(int)((r-l)/2+1), r);
    }
  }
}
```

```java
/** Benchmark:  1 Objekt erzeugen und darin einen rekursive
  * Abstieg ausfuehren.
  */

public class RecBenchmark extends Benchmark {

  public void runBenchmark() {
    Rec a = new Rec(1000001);
    a.randomFill();
    beginBenchmark();                   // Test-Start
    double d = a.doRec(0, 1000000);     // Rekursion aufrufen
    endBenchmark();                     // Test-Ende
  }

  public void infoBenchmark() {
    printBenchmark("SMall JVM-Benchmark-Collection:  "
                   + "rekursiver Abstieg");
    printBenchmark("v0.1");
    printBenchmark("Marc Schanne "
                   + <Marc.Schanne@stud.uni-karlsruhe.de>\n");
  }

  /** Main-Prog.  fuer Application */

  public static void main(String argv[]) {
    RecBenchmark pb = new RecBenchmark();
    pb.infoBenchmark();
    pb.runBenchmark();
  }
}
```

Anhang B

Meßwerte der Benchmarks

SPARCStaion 4 - Solaris

	JDK 1.0.2	**JDK 1.0.2 dp11/13/96**	**Netscape 3.01**	**Kaffe 0.7.0**
Matrizen Multiplikation	95.206	(100.525)	95.751	(23.092)
	(94.245)	98.993	(95.746)	(18.319)
	94.919	(98.094)	(97.013)	19.137
	(95.867)	99.311	96.417	18.481
	95.063	**99.152**	**96.084**	**18.809**
	100.0%	**104.3%**	**101.1%**	**19.8%**
Gauß Elimination	(163.736)	(157.122)	(199.187)	(75.009)
	167.294	158.525	196.221	(59.765)
	(167.607)	(159.879)	194.445	59.855
	166.663	159.528	(193.489)	59.804
	166.979	**159.027**	**195.333**	**59.830**
	100.0%	**95.2%**	**117.0%**	**35.8%**
Polynom Multiplikation	(199.064)	(202.198)	(229.672)	(438.695)
	(173.273)	(185.342)	234.463	(414.036)
	176.199	187.126	(281.229)	415.763
	177.353	186.543	235.605	416.213
	176.776	**186.835**	**235.034**	**415.988**
	100.0%	**105.7%**	**133.0%**	**235.3%**
Primezahlen Objekt erzeugen	(30.996)	24.872	(35.114)	(8.058)
	30.778	24.454	(44.902)	(6.388)
	29.710	(24.398)	38.746	6.507
	(29.698)	(25.531)	41.130	6.393
	30.244	**24.663**	**39.938**	**6.450**
	100.0%	**81.5%**	**132.1%**	**21.3%**

Fortsetzung der Tabelle von Seite 66:

	JDK 1.0.2	JDK 1.0.2 dp11/13/96	Netscape 3.01	Kaffe 0.7.0
Sieb	91.420	(83.746)	(95.369)	(17.047)
	91.643	(86.861)	(90.690)	(11.026)
	(91.614)	86.170	91.618	11.086
	(92.802)	84.835	93.243	11.112
	91.532	**85.503**	**92.431**	**11.099**
	100.0%	**93.4%**	**101.0%**	**12.1%**
Sortieren	(200.743)	(167.288)	201.472	(27.400)
	182.615	170.002	(201.663)	27.317
	184.453	170.292	196.070	(26.065)
	(181.285)	(170.710)	(193.222)	26.325
	183.534	**170.147**	**198.771**	**26.821**
	100.0%	**92.7%**	**108.3%**	**14.6%**

PC - Windows 95

	JDK 1.0.2 ss08/12/96	Super-Cede 1.0b	Visual Cafe 2.00b07	Netscape 3.01	IExplorer 3.02
Matrizen Multiplikation	(67.72) 67.78 67.83 (68.94) **67.81** **100.0%**	(13.78) 14.06 13.90 (14.11) **13.98** **20.6%**	(7.96) 7.86 7.91 (7.90) **7.89** **11.6%**	(11.37) 11.15 (10.99) 11.10 **11.13** **16.4%**	(8.56) 8.07 (7.97) 8.08 **8.08** **11.9%**
Gauß Elimination	145.72 (145.72) 146.08 (146.46) **145.90** **100.0%**	15.77 15.87 (15.82) (15.60) **15.82** **10.8%**	17.79 17.79 (17.79) (17.77) **17.79** **12.2%**	18.23 (18.34) (18.13) 18.24 **18.24** **12.4%**	(14.88) 14.83 (14.83) 14.85 **14.84** **10.2%**
Polynom Multiplikation	109.57 (113.14) (108.31) 109.92 **109.75** **100.0%**	142.42 143.30 (142.09) (144.29) **142.86** **130.2%**	(51.13) 50.58 50.81 (50.47) **50.70** **46.2%**	223.49 (221.62) (223.66) 223.55 **223.52** **203.7%**	118.69 (118.48) 118.69 (118.86) **118.69** **108.1%**
Primzahlen Objekt erzeugen	(24.39) 23.89 23.84 (23.67) **23.87** **100.0%**	2.03 (2.04) 2.03 (2.03) **2.03** **8.5%**	2.69 (2.47) 2.64 (2.74) **2.67** **11.2%**	(4.91) 6.43 6.54 (8.51) **6.49** **27.2%**	(2.42) (3.35) 2.88 2.96 **2.92** **12.2%**
Sieb	79.97 79.97 (79.98) (79.82) **79.97** **100.0%**	(4.12) 4.17 4.17 (4.17) **4.17** **5.2%**	6.20 6.16 (6.20) (6.10) **6.18** **7.7%**	4.51 4.51 (4.45) (4.51) **4.51** **5.6%**	4.12 4.12 (4.06) (4.12) **4.12** **5.2%**
Sortieren	147.59 (146.15) (148.96) 147.63 **147.61** **100.0%**	41.90 (41.96) (41.08) 41.91 **41.91** **28.4%**	9.94 (10.00) 9.77 (8.84) **9.86** **6.7%**	(20.71) (20.21) 20.26 20.27 **20.27** **13.7%**	(10.82) 10.99 10.88 (11.04) **10.94** **7.4%**

PC - Linux 2.0.x

	JDK 1.0.2 chapman	Netscape 3.02	Kaffe 0.7.0
Matrizen Multiplikation	93.589	99.401	17.442
	(93.613)	(98.849)	17.402
	93.573	100.250	(17.621)
	(93.446)	(101.384)	(17.255)
	93.581	**99.826**	**17.422**
	100.0%	**106.7%**	**18.6%**
Gauß Elimination	179.399	(201.751)	(55.888)
	(179.649)	201.451	55.822
	(178.983)	(201.387)	(55.681)
	179.236	201.445	55.806
	179.318	**201.448**	**55.814**
	100.0%	**112.3%**	**31.1%**
Polynom Multiplikation	144.901	(218.059)	335.903
	(144.973)	223.024	(*)
	(144.570)	224.670	337.378
	144.800	(225.816)	(342.335)
	144.851	**223.847**	**336.641**
	100.0%	**154.5%**	**232.4%**
Primzahlen Objekt erzeugen	(35.012)	28.735	(3.463)
	35.034	(29.706)	3.472
	35.090	28.182	3.489
	(39.381)	(28.105)	(3.544)
	35.062	**28.459**	**3.481**
	100.0%	**81.2%**	**9.9%**
Sieb	128.143	(106.365)	(7.620)
	(128.090)	107.213	7.625
	128.697	(108.195)	7.646
	(147.805)	106.467	(7.744)
	128.420	**106.840**	**7.636**
	100.0%	**83.2%**	**5.9%**
Sortieren	202.869	(197.112)	(13.921)
	(202.852)	(196.913)	13.886
	(203.439)	197.066	(13.844)
	203.239	197.109	13.872
	203.054	**197.088**	**13.879**
	100.0%	**97.1%**	**6.8%**

* java.lang.NullPointerException

Anhang C

Verwendete Abkürzungen

API	Application Programming Interface
AWT	Abtract Window(ing) Toolkit
GNU	GNU's Not Unix
GUI	Graphic User Interface
IE	Microsoft Internet Explorer
JDK	Java Development Kit
JIT	Just in Time
JITC	Just in Time-Compiler
JVM	Java Virtual Machine
LGS	Lineares Gleichungssystem
MFC	Microsoft Foundation Classes
OMT	Object Modeling Technique
O(n^3)	Komplexitätsmaß für kubische Laufzeit
URL	Uniform Resource Locator
WWW	World Wide Web
X	X-Window-System

Literaturverzeichnis

[BISS97] The BISS AWT framework. Version 0.86. Wilhelmshaven: BISS GmbH, Januar 1997.

[Dalh97] MATTHIAS KALLE DALHEIMER: Java Virtual Machine. Sprache, Konzept, Architektur. Köln: O'Reilly Verlag, 1997.

[Dong96] JACK J. DONGARRA: Performace of Various Computers using Standard Linear Equations Software. Computer Science department, University of Tennessee, November 1996. http://www.netlib.org/benchmark/performance.ps

[Flan96] DAVID FLANAGAN: Java in a Nutshell. A desktop Quick Reference for Java Programmers. Sebastopol: O'Reilly & Associates, Inc., 1996.

[Foot96] BILL FOOTE: Integrating Java with C++. Learn how to use C++ code from within a Java application and how to call from C++ to a Java object. Oktober 1996. http://www.javaworld.com/javaworld/javatips/jw-javatip17.html

[GHJV96] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, JOHN VLISSIDES: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. Bonn: Addison-Wesley, 1996. S. 401ff.

[Günt97] EDWIN GÜNTHNER: Portierung der JVM auf den Video- und Multimediaprozessor TMS320C80. Studienarbeit am Institut für Programmstrukturen und Datenorganisation. Karlsruhe, 1997.

[Java96a] The Java Native Code API. JavaSoft, 30.7.1996. http://www.javasoft.com/doc/jit_interface.html

[Java96b] The Java Native Method Interface. JavaSoft, 1996. http://www.javasoft.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html

[LiYe96] TIM LINDHOLM, FRANK YELLIN: The Java Virtual Machine Specifikation. Addison-Wesley, 1996.

[Phil96] MICHAEL PHILIPPSEN (Hg.): Java Seminarbeiträge. Technical report 24/96. Universität Karlsruhe, Juli 1996.

[Sedg91.1] ROBERT SEDGWICK: Algorithmen. Bonn: Addison-Wesley, 1991. S. 607ff.

[Sedg91.2] ebd. S. 591ff.

[Sedg91.3] ebd. S. 136ff.

[Sun94] The Java Language: A White Paper. Sun Microsystems, 1994.

[Sun95a] The Java Language Specification. Version 1.0 Beta. Sun Microsystems, Oktober 1995.

[Sun95b] The Java Virtual Machine Specification. Sun Microsystems, März 1995.

[Sun95c] The Java Language Environment. A White Paper. Sun Microsystems, Mai 1995.

[Venn96] BILL VENNERS: Unter the Hood: Java's garbage-collected heap. An inroduction to the garbage-collected heap of the HJava virtual machine. Juli 1996.
http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html

[Yu95] NELSON YU: Inroduction to the AWT. 1995.