CACAO - Eine effiziente JavaVM Implementierung

Andreas Krall

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien
http://www.complang.tuwien.ac.at/andi/

Zusammenfassung CACAO ist eine effiziente Implementierung einer JavaVM, die auf Just-In-Time Übersetzung beruht. CACAO übersetzt während der Laufzeit die aufgerufenen Methoden auf Maschinencode für den Alpha Prozessor. Der Übersetzer formt dabei den stackbasierten Zwischencode in den registerbasierten Maschinencode von RISC-Prozessoren um. Dabei werden Befehle kombiniert, Kopierbefehle entfernt und den Variablen Maschinenregister zugewiesen. Bei der Entwicklung des Übersetzers wurde auf extrem kurze Übersetzungszeit und geringen Speicherverbrauch geachtet. CACAO ist zur Zeit die schnellste JavaVM Implementierung für den Alpha-Prozessor und benötigt weniger als 1000 Zyklen für die Übersetzung eines Zwischencodebefehls.
Zur Zeit wird CACAO um Maschinencodegeneratoren für den MIPS-, PowerPC- und Sparc-Prozessor erweitert. Gleichzeitig dazu werden die Möglichkeiten geschaffen, den erzeugten Maschinencode abzuspeichern, und nicht nur Just-In-Time, sondern das ganze Programm auf einmal, auch für andere Zielarchitekturen zu übersetzen.

1 Einleitung

Java's [AG96] Erfolg als Programmiersprache kommt von seiner Rolle als Internet-Programmiersprache. Die Grundlage für diesen Erfolg ist die maschinenunabhängige Darstellung von Programmen im Zwischencode der JavaVM [LY96]. Die Standardimplementierung der JavaVM als Interpreter macht die Ausführung von Programmen langsam. Das spielt keine Rolle, wenn kleine Anwendungen in einem Browser ausgeführt werden, ist aber bei großen Programmen oder zeitkritischen Anwendungen inakzeptabel. Zwei Lösungen für dieses Problem bieten sich an:

- spezialisierte Java-Prozessoren
- Übersetzung des Zwischencodes auf Maschinencode

SUN hat beide Wege eingeschlagen und entwickelt sowohl Java-Prozessoren als auch Übersetzer. Mit CACAO haben wir uns für Übersetzer entschieden, da dieser Ansatz portabler ist und mehr Möglichkeiten für Optimierungen bietet

[KEG98]. Die Übersetzung auf Maschinencode kann auf zwei Arten durchgeführt werden: Übersetzung des gesamten Programms auf einmal oder Übersetzung der aufgerufenen Methoden während der Laufzeit (Just-In-Time Übersetzer). CACAO unterstützt beide Methoden. CACAO und seine Implementierung sind in mehreren Artikeln beschrieben [KG97] [KP98] und ist über das Internet frei erhältlich.

2 Verwandte Arbeiten

Die Idee maschinenunabhängiger Zwischendarstellungen ist schon sehr alt und geht bis in das Jahr 1960 zurück [Ste61]. Die Zwischensprache UNCOL (UNiversal Computer Oriented Language) wurde für eine Verwendung in Übersetzern vorgeschlagen, um das Problem der Übersetzung vieler verschiedener Programmiersprachen auf unterschiedliche Architekturen zu lösen.

Das Design der JavaVM wurde sehr stark von P-Code, der virtuellen Maschine von Pascal [PD82], beeinflußt. P-Code erlangte großen Bekanntheitsgrad durch seine Verwendung im UCSD-Pascal-System. Genauso wie bei der JavaVM gab es Mikroprozessoren, die P-Code direkt ausführen konnten. Am Markt waren diese Prozessoren aber nicht erfolgreich.

Das Amsterdam-Compiler-Kit [TvSKS83] [TKLJ89] verwendet eine stackbasierte Zwischendarstellung. Diese Darstellung wurde entworfen, um trotz kurzer Übersetzungszeiten schnellen Maschinencode erzeugen zu können. Die Zwischendarstellung des Gardens Point Übersetzers beruht auch auf einer Stackmaschine, die *Dcode* genannt wurde [Gou97]. *Dcode* wurde stark von P-Code beeinflußt. Für *Dcode* wurden sowohl Interpreter als auch Übersetzer für verschiedene Architekturen entwickelt.

Die Probleme, die bei der Übersetzung von Stackcode nach Maschinencode auftreten, sind auch von der Programmiersprache Forth bekannt. In seiner Doktorarbeit [Ert96] und in [EP97] beschreibt Anton Ertl RAFTS, ein Forth System, das Maschinencode während der Laufzeit erzeugt. Die Übersetzung in Maschinencode erfolgt über einen Zwischenschritt, in welchem die Stackbefehle wieder in arithmetische Ausdrücke zurückübersetzt werden, die in einem azyklischen gerichteten Graphen dargestellt werden. In [EM95] übersetzt Ertl die Programmiersprache Forth unter Verwendung von C auf Maschinencode. Dabei werden Stackzellen durch lokale Variablen einer C-Funktion dargestellt. Optimierung und Maschinencodeerzeugung werden dann vom C-Übersetzer durchgeführt.

Die ersten JIT Übersetzer kamen vor zwei Jahren von SUN und für die Browser von Netscape und Microsoft heraus. Darauf folgte bald die Entwicklungsumgebung von Symantec. SUN brachte auch JIT Übersetzer für Sparc- und PowerPC-Prozessoren heraus, SGI für MIPS-Prozessoren und vor kurzem Digital für den Alpha-Prozessor. Kaffe ist ein frei verfügbarer JIT Übersetzer der von Tim Wilkinson entwickelt wurde (`http://www.kaffe.org/`). Für alle diese erwähnten Systeme gibt es keine Beschreibung der verwendeten Übersetzungstechniken.

Das Übersetzungsschema von *Caffeine* wird im Artikel von Hsieh und seinen Kollegen beschrieben [HGH96]. Es werden dabei zwei Übersetzungmethoden verwendet. Die einfache Methode emuliert den Stack, die bessere Methode ersetzt den Stack durch Register. *Caffeine* ist kein JIT Übersetzer, es übersetzt ein komplettes Programm im vorhinein. DAISY (Dynamically Architected Instruction Set from Yorktown) ist eine VLIW Architektur die bei IBM entwickelt wurde, um eine effiziente Ausführung von PowerPC, S/390 und JavaVM Maschinencode zu ermöglichen. Die Kompatibiliät mit alten Architekturen wird dabei durch JIT Übersetzung ermöglicht. Der Übersetzer für die JavaVM ist in [EAH97] beschrieben.

Adl-Tabatabai und seine Kollegen [ATCL+98] beschreiben eine effiziente Methode der Maschinencodeerzeugung für einen JIT Übersetzer. Dieser Übersetzer macht Optimierungen wie Elimination von Bereichsüberprüfungen, Auswertung von gemeinsamen Teilausdrücken und zwei verschiedene Arten von Registerzuteilung, eine einfache und eine globale auf Prioritäten basierte. Die Ergebnisse zeigen, daß für viele Benchmarkprogramme die komplexe Registerzuteilung mehr Übersetzungszeit verbraucht als durch kürzere Laufzeit wieder eingebracht wird.

3 Übersetzungsgrundlagen

Die JavaVM ist eine getypte Stackmaschine [LY96] mit unterschiedlichen Befehlen für ganze Zahlen, Fließkommazahlen und Zeiger. Der Befehlsatz besteht aus arithmetisch/logischen Befehlen, Vergleichs- und Sprungbefehlen, Befehlen für Variablen- und Speicherzugriff, Methodenaufruf, Typüberprüfung und Synchronisation. Das folgende Beispiel zeigt die Zwischendarstellung für die Java-Zuweisung `a = b - c * d`.

```
iload b   ; push contents of variable b
iload c   ; push contents of variable c
iload d   ; push contents of variable d
imul      ; compute c * d
isub      ; compute b - (c * d)
istore a  ; pop stack top into variable a
```

Die Architektur von RISC-Prozessoren unterscheidet sich wesentlich von der Stackarchitektur der JavaVM. RISC-Prozessoren haben einen großen Registersatz (der Alpha-Prozessor besitzt 32 Ganzzahl- und 32 Fließkommazahlregister, welche alle 64 Bit breit sind). Arithmetisch/logische Befehle werden nur auf Daten ausgeführt, die sich in Registern befinden. Speicherbefehle bewegen die Daten zwischen Speicher und Register. Die lokalen Variablen einer Methode befinden sich normalerweise in Registern und werden nur in den Speicher ausgelagert, wenn zuwenig Register vorhanden sind oder Aufrufvorschriften es verlangen.

Die vorhergehende Javazuweisung `a = b - c * d` würde von einem optimierenden Übersetzer in die folgenden zwei Alpha-Maschinenbefehle übersetzt werden (die Variablen `a`, `b`, `c` und `d` befinden sich in Registern):

```
MULL c,d,tmp0   ; tmp0 = c * d
SUBL b,tmp0,a   ; a = b - tmp0
```

Wenn JavaVM Zwischencode auf Maschinencode übersetzt wird, wird der Stack vollständig eliminiert und die einzelnen Stackzellen werden durch temporäre Variablen ersetzt, die sich normalerweise in Registern befinden. Eine naive Übersetzung des vorhergehenden Beispiels würde folgende Alpha-Befehle erzeugen:

```
MOVE b,t0       ; iload b
MOVE c,t1       ; iload c
MOVE d,t2       ; iload d
MULL t1,t2,t1   ; imul
SUBL t0,t1,t0   ; isub
MOVE t0,a       ; istore a
```

Um JavaVM-Befehle auf Maschinencode abzubilden, ist daher eine effiziente Methode zum Entfernen oder Vermeiden der Kopierbefehle nötig. Weiters muß ein schneller Algorithms für die Registerzuteilung angewendet werden.

4 Der CACAO Übersetzer

Die Übersetzung erfolgt in drei Durchgängen. Im ersten Durchgang werden die Grundblöcke bestimmt und es wird eine Zwischendarstellung der JavaVM-Befehle aufgebaut, die einfacher zu Dekodieren ist. Im zweiten Durchgang wird der Stack analysiert, wobei eine Datenstruktur erzeugt wird, die einer statischen Stackdarstellung entspricht. Gleichzeitig werden dabei Befehle kombiniert, Variablenabhängigkeiten überprüft und der Registerbedarf bestimmt. Im letzten Durchgang werden gleichzeitig mit der Maschinencodeerzeugung auch die temporären Register zugeteilt.

4.1 Grundblockschnittstellendefinition

Eine optimale Registerzuteilung bei der Vereinigung von Kontrollflüssen erfordert aufwendige Verfahren. Wir haben daher eine fixe Schnittstellendefinition an den Grenzen von Grundblöcken festgesetzt. Jede Stackzelle an einer Grundblockgrenze wird einem fixen Register zugewiesen. Während der Stackanalyse wird dabei der Typ des Registers bestimmt und ob der Inhalt des Registers über Methodenaufrufe erhalten bleiben muß. Um die Größe von Grundblöcken zu erhöhen, unterbrechen Methodenaufrufe nicht einen Grundblock. Um die Auswirkungen dieser fixen Registerschnittstelle zu untersuchen, haben wir eine Analyse an einer großen Java-Anwendung, dem `javac` Übersetzer und allen seinen Bibliotheksmethoden, durchgeführt. Tabelle 1 zeigt, daß in mehr als 93% aller Fälle der Stack an Grundblockgrenzen leer ist und die maximale Stacktiefe 6 ist. Damit wird klar, daß diese fixe Registerzuteilung keinen negativen Einfluß auf die Qualität des erzeugten Maschinencodes hat.

Stacktiefe	0	1	2	3	4	5	6	>6
Schnittstellen	7930	258	136	112	36	8	3	0

Tabelle1. Verteilung der Stacktiefe an Grundblockgrenzen

4.2 Entfernen von Kopierbefehlen

Um unnötige Kopierbefehle zu entfernen, wird das Laden von Daten solange verzögert, bis die Instruktion erreicht wird, die diese Daten benötigt. Abbildung 1 zeigt die Befehls- und Stackdarstellung des Übersetzers. Dieser Stack ist nicht der Laufzeitstack sondern eine statische Darstellung des Stackzustandes, wie er während der Abarbeitung des Programmes auftreten würde.

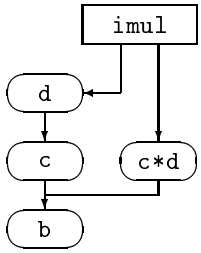

Abbildung1. Befehls- und Stackdarstellung

Ein Befehl hat Zugriff auf den Stack vor und nach der Ausführung des Befehls. Der Stack wird dabei durch eine lineare Liste dargestellt. Die beiden Stacks können dabei als die Quelloperanden und der Zieloperand des Befehls gesehen werden. Diese Darstellung kann nun einfach für die Vermeidung von Kopierbefehlen verwendet werden. Jede Stackzelle enthält nicht nur den Typ dieser Zelle, sondern auch Informationen darüber von welcher lokalen Variablen diese Zelle eine Kopie ist. Kopierbefehle von Variablen an Stackzellen werden nicht erzeugt. Der Befehl, der diese Zelle als Quelloperand benötigt, verwendet direkt die lokale Variable.

Probleme mit dieser Methode können bei `store`-Befehlen auftreten. Im Beispiel von Abb. 2 enthält der Stack ein Kopie der lokalen Variablen `a`. Der Befehl `istore a` schreibt einen neuen Wert in die Variable `a` und macht damit eine spätere Verwendung dieser Variablen ungültig. Um das zu Vermeiden muß die lokale Variable auf eine Stackzelle (Register) kopiert werden und diese Kopie verwendet werden.

Da der Stack als lineare Liste dargestellt ist, muß bei `store`-Befehlen die-

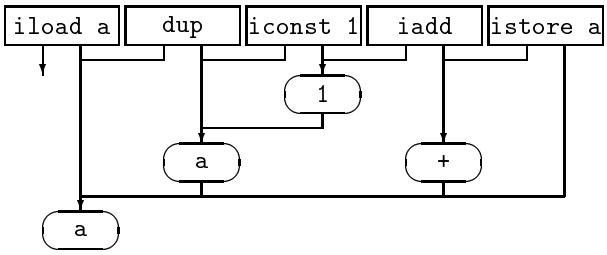

Abbildung2. Abhängigkeiten

se Liste nach Kopien von lokalen Variablen durchsucht werden. Tabelle 2 gibt die Verteilung der Stacktiefe bei `store`-Befehlen an. Die Stacktiefe ist hier fast immer 0.

stack depth	0	1	2	3	>3
occurrences	2167	31	1	3	0

Tabelle2. Verteilung der Stacktiefe bei `store`-Befehlen

Diese statische Stackdarstellung wird auch dazu verwendet, um Informationen von `store`-Befehlen an Befehle zurückzubringen, die ein Ergebnis berechnen und es in einer Stackzelle ablegen würden. Dabei gibt es ähnliche Probleme mit Abhängigkeiten zu lösen.

4.3 Registerzuteilung

Aufwendige Registerzuteilungsalgorithmen sind weder angebracht noch nötig. Der `javac` Übersetzer teilt verschiedenen lokalen Variablen die selbe Variablennummer zu, wenn sich die Lebensbereiche dieser Variablen nicht überlappen. Stackzellen haben ihren Lebensbereich implizit durch ihre Stackposition gekennzeichnet. Während der Stackanalyse werden Stackzellen gekennzeichnet, deren Inhalt über einen Methodenaufruf hinweg erhalten bleiben müssen. Diesen Stackzellen und lokalen Variablen werden dann gesicherte Register zugeteilt.

Ein schneller Methodenaufruf ist für die Geschwindigkeit von Java wichtig. Daher werden Argumentregister und das Rückgaberegister von Methoden ähnlich behandelt wie `store`-Befehle.

4.4 Kombination von Befehlen

Zusammen mit der Stackanalyse werden Befehle, die Konstanten laden, mit ausgewählten Befehlen kombiniert, die direkt darauf folgen. In diese Klasse fallen Befehle wie `iadd`, `isub`, `imul`, `idiv`, logische und Verschiebebefehle und Vergleichs- und Sprungbefehle. Während der Maschinencodeerzeugung wird überprüft, ob der Wert der Konstanten im gültigen Bereich für Maschinenbefehle liegt und entsprechender Maschinencode erzeugt.

4.5 Beispiel

Das Beispiel in Abbildung 3 zeigt die Zwischendarstellung des Übersetzers wie sie für Fehlersuche verwendet wird. `Local Table` gibt die Typen und die Registerzuteilung von lokalen Variablen an. Der Java-Übersetzer verwendet die selbe Variablennummer für unterschiedliche lokale Variablen, wenn sich die Gültigkeitsbereiche nicht überlappen. In diesem Beispiel wird die Variablennummer 3 sogar für Variablen unterschiedlichen Typs genutzt (ganze Zahl und Adresse). Der JIT-Übersetzer hat das gesicherte Register mit der Nummer 12 dieser Variablen zugeteilt.

Für den Grundblock mit dem Namen `L004` wird in diesem Beispiel ein Interfaceregister benötigt. Am Beginn des Grundblocks wird das Interfaceregister in das Argumentregister `A00` kopiert. Dies ist einer der seltenen Fälle, wo ein aufwendigerer Algorithmus zum Entfernen der Kopierbefehle ein Argumentregister für das Interface verwenden hätte können.

Bei den Befehlen 2 und 3 erkennt man die Kombination einer Konstanten mit einem Rechenbefehl. Da die Befehle in einem Feld gespeichert sind, muß die freie Zelle mit einem Leerbefehl überschrieben werden. Der Befehl `IADDCONST` hat bereits die lokale Variable `L02` als Ziel, eine Information die vom nachfolgenden Befehl `ISTORE` kommt. In ähnlicher Weise hat der Befehl `INVOKESTATIC` mit der Nummer 31 alle seine Operanden als Argumentregister markiert. In diesem Beispiel werden alle Kopierbefehle (mit Ausnahme des Interfaceregisters) entfernt.

4.6 Just-In-Time Übersetzer

Zu Beginn werden alle Zeiger in der Methodentabelle mit Zeigern auf den Übersetzer initialisiert. Wird dann der Übersetzer aufgerufen, wird der erzeugte Maschinencode in den Speicher geschrieben und danach die neu erzeugte Methode aufgerufen.

Um die Fehlersuche im CACAO-Übersetzer zu erleichtern, gibt es einen Modus, in dem der Übersetzer Maschinencode für alle Methoden erzeugt und diese auf eine Datei ausgibt, ohne die Methoden aufzurufen. Dieser Modus ist auch geeignet, um Maschinencode für eine andere Zielarchitektur zu erzeugen. Gegenwärtig arbeiten wir daran, die Maschinencodegeneratoren für MIPS-, PowerPC- und Sparc-Prozessoren zu entwickeln und die Unterstützung für eingebettete Systeme zu verbessern.

4.7 Darstellung von Objekten

Die Darstellung von Objekten und Klassen wurde so entworfen, daß ein schneller Zugriff auf Objekte und Methoden bei geringem Speicherverbrauch möglich ist. In CACAO enthalten Objekte die Instanzvariablen und einen Zeiger auf die Klasse. Für die Klassendarstellung gibt es zwei Varianten. In der kompakteren Variante enthält eine Klasse die Zeigertabelle der virtuellen Funktionen und mit negativem Offset die Interfacetabelle (Abbildung 4). Für jedes implementierte Interface existiert ein Zeiger auf die virtuelle Funktionstabelle der Interfaces.

In der schnelleren Variante werden Interfacemethoden, das sind alle Methoden, die in Interfaces verwendet werden, mit negativem Offset zum Klassenzeiger gespeichert (Abbildung 5). Durch die Unterscheidung in Interfacemethoden und virtuelle Methoden bleibt die normale Zeigertabelle unverändert und auch die Interfacemethodentabelle wird kleiner. Trotzdem ist der Speicherverbrauch für die Interfacemethodentabelle quadratisch (Anzahl der Klassen mit Interfaces * Anzahl der Interfacemethoden). Mittels *selector coloring* kann die Anzahl der unterschiedlichen Offsets für Interfacemethoden und damit der Speicherverbrauch stark reduziert werden. Werden Klassen dynamisch geladen, dann hat diese Methode allerdings den Nachteil, daß sich die Offsets unter Umständen ändern können und daher Änderungen im bereits erzeugten Maschinencode notwendig werden.

5 Ergebnisse

Um die Geschwindigkeit von CACAO zu testen, haben wir CACAO mit einer Portierung von Sun's JDK Interpreter, dem JIT-Übersetzer kaffe und dem JIT-Übersetzer von Digital verglichen (siehe Tabelle 3. Digitals System hat Probleme mit der Synchronisation (siehe auch [KP98]), daher sind hier die Zeiten dreimal langsamer als bei Sun's Interpreter.

Tabelle 4 vergleicht CACAO mit einem C Übersetzer. Die Option `-cb` von CACAO schaltet die Bereichsüberprüfung aus. Da C auch keine Bereichsüberprüfungen durchführt, ist dieser Vergleich fair.

6 Zusamenfassung

Wir haben CACAO und seinen Übersetzer beschrieben. CACAO ist die zur Zeit schnellste JavaVM-Implementierung für den Alpha-Prozessor. CACAO wird um Codegeneratoren für andere Prozessoren erweitert und hat Unterstützung für eingebettete Systeme. CACAO kann via WWW bezogen werden:
`http://www.complang.tuwien.ac.at/java/cacao/`.

7 Danksagung

Ich danke Reinhard Grafl und Mark Probst für Korrekturen dieses Artikels.

	JavaLex	javac	espresso	Toba	java_cup
run time on 21064A 300MHz (in seconds)					
JDK	29.8	18.5	8.7	32.1	3.5
kaffe	9.9	17.8	12.5	-	2.98
Digital Interpr.	98.9	56.4	17.9	-	9.2
Digital JIT	84.4	47.6	14.1	-	9.8
CACAO	2.08	2.92	1.88	3.85	0.50
number of compiled JavaVM instructions					
	13412	34759	27281	14430	17489
speedup with respect to interpreter					
speedup JDK/kaffe	3.01	1.04	0.7	-	1.17
speedup JDK/DEC-JIT	0.35	0.38	0.48	-	0.36
speedup JDK/CACAO	12.79	4.87	3.85	7.77	4.02

Tabelle3. Vergleich von JDK, kaffe, Digitals JIT und CACAO

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[ATCL+98] Ali-Reza Adl-Tabatabai, Michal Ciernak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Conference on Programming Language Design and Implementation*, volume 33(6) of *SIGPLAN*, page to appear, Montreal, 1998. ACM.

[EAH97] Kemal Ebcioğlu, Erik Altman, and Erdem Hokenek. A Java ILP machine based on fast dynamic compilation. In *MASCOTS'97 - International Workshop on Security and Efficiency Aspects of Java*, 1997.

[EM95] M. Anton Ertl and Martin Maierhofer. Translating Forth to native C. In *EuroForth '95*, 1995.

[EP97] M. Anton Ertl and Christian Pirker. The structure of a Forth native code compiler. In *EuroForth '97 Conference Proceedings*, pages 107–116, 1997.

[Ert96] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, April 1996.

[Gou97] K. John Gough. Multi-language, multi-target compiler development: Evolution of the Gardens Point compiler project. In Hanspeter Mössenböck, editor, *JMLC'97 –Joint Modular Languages Conference*, Linz, 1997. LNCS 1204.

[HGH96] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen-mei W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'29)*, 1996.

[KEG98] Andreas Krall, Anton Ertl, and Michael Gschwind. JavaVM implementations: Compilers versus hardware. In John Morris, editor, *Australian*

	sieve	addition	linpack
run time on 21064A 300MHz (in seconds)			
JDK	83.2	138.76	1.6
Digital interpr.	70.3	124.4	2.0
kaffe	9.14	12.2	0.34
Digital JIT	6.27	5.33	0.9
GCC -O3	2.0	1.40	-
CACAO total	4.57	1.69	0.81
CACAO -cb runtime only	3.31	1.42	0.13
relation of run time			
speedup JDK/kaffe	9.10	11.8	4.7
speedup JDK/DEC-JIT	13.3	29.2	1.7
speedup JDK/CACAO	18.2	82.1	2.0
speedup JDK/CACAO -cb	24.1	85.7	4.8
speedup JDK/GCC	41.6	99.1	-
CACAO -cbnf run/GCC	1.66	1.01	-

Tabelle4. Vergleich von JDK, kaffe, Digitals JIT und CACAO

Computer Architecture Conference (ACAC'98), volume 20(4) of *Australian Computer Science Communications*, pages 101–110, Perth, 1998. Springer.

[KG97] Andreas Krall and Reinhard Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[KP98] Andreas Krall and Mark Probst. Monitors and exceptions: How to implement Java efficiently. In Siamak Hassanzadeh and Klaus Schauser, editors, *ACM 1998 Workshop on Java for High-Performance Computing*, pages 15–24, Palo Alto, March 1998. ACM.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[PD82] Steven Pemberton and Martin C. Daniels. *Pascal Implementation, The P4 Compiler*. Ellis Horwood, 1982.

[Ste61] T. B. Steel. A first version of UNCOL. In *Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference*, pages 371 – 377, 1961.

[TKLJ89] A. S. Tanenbaum, M. F. Kaashoek, K. G. Langendoen, and C. J. H. Jacobs. The design of very fast portable compilers. *ACM SIGPLAN Notices*, 24(11):125–131, November 1989.

[TvSKS83] Andrew S. Tanenbaum, Hans van Staveren, E. G. Keizer, and Johan W. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 16(9):654–660, September 1983.

This article was processed using the LaTeX macro package with LLNCS style

```
java.io.ByteArrayOutputStream.write (int)void

Local Table:        0:                (adr) S15
                    1:    (int) S14
                    2:    (int) S13
                    3:    (int) S12   (adr) S12

Interface Table:    0:    (int) T24

[                 L00]        0  ALOAD         0
[                 T23]        1  GETFIELD      16
[                 L02]        2  IADDCONST     1
[                 L02]        3  NOP
[                    ]        4  ISTORE        2
[                 L02]        5  ILOAD         2
[             L00 L02]        6  ALOAD         0
[             T23 L02]        7  GETFIELD      8
[             T23 L02]        8  ARRAYLENGTH
[                    ]        9  IF_ICMPLE     L005
                               ...............

[                    ]       18  IF_ICMPLT     L003
[                    ] L002:
[                 I00]       19  ILOAD         3
[                 I00]       20  GOTO          L004
[                    ] L003:
[                 I00]       21  ILOAD         2
[                 A00] L004:
[                 L03]       22  BUILTIN1      newarray_byte
[                    ]       23  ASTORE        3
[                 L00]       24  ALOAD         0
[                 A00]       25  GETFIELD      8
[             A01 A00]       26  ICONST        0
[         A02 A01 A00]       27  ALOAD         3
[     A03 A02 A01 A00]       28  ICONST        0
[ L00 A03 A02 A01 A00]       29  ALOAD         0
[ A04 A03 A02 A01 A00]       30  GETFIELD      16
[                    ]       31  INVOKESTATIC  java.lang.System.arraycopy
[                 L00]       32  ALOAD         0
[             L03 L00]       33  ALOAD         3
[                    ]       34  PUTFIELD      8
[                    ] L005:
                               ...............

[                    ]       45  RETURN
```

Abbildung3. Beispiel: Befehls- und Stackdarstellung

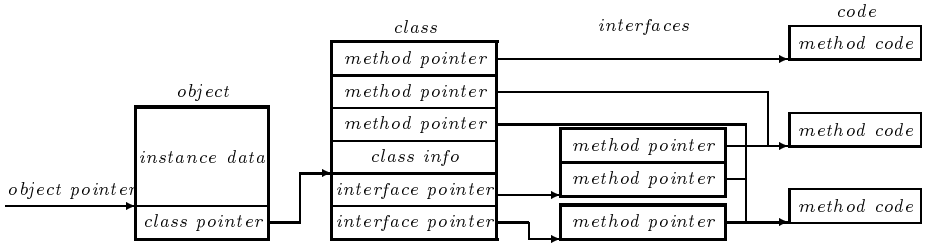

Abbildung4. kompakte Objekt- und Klassendarstellung

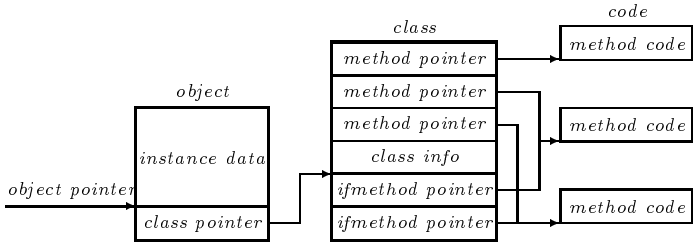

Abbildung5. schnelle Objekt- und Klassendarstellung