CACAOL -- From the fastest JIT to JVM

Andreas Krall

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien

andi@complang.tuwien.ac.at
Contents

Introduction

The Java Virtual Machine

Translation into register orientationed representation

Other implementation details of CACAO

Some results
System architecture of CACAO

Java program

loade

JIT compiler

native methods

GC

class library

operating system

prozessor
The Java Virtual Machine

- stack architecture
- typed
- checked
- class files
- constant pool
Stack Architecture  Register Architecture

\[ a = b \times c + d \]

- iload b
- iload c
- imul
- iload d
- iadd
- istore a
- mull b,c,t0
- addl t0,d,a
Naive translation into native code

\[ a = b \times c + d \]

iloadd \( b \)  \( b \)

iloadd \( c \)  \( b \) \( c \)
imul \( \times \)

iloadd \( d \)  \( \times \) \( d \)
iadd \( + \)

istore \( a \)

mov \( b, t0 \)

mov \( c, t1 \)
mull \( t0, t1, t0 \)

mov \( d, t1 \)

addl \( t0, t1, t0 \)

mov \( t0, a \)

stack element \( \rightarrow \) stack register
CACAO - Compiler

4+ pass compiler

• determines basic blocks
• analyses stack
• allocates registers
• generates native code
• optional loop optimizations and verification
Static stack representation

- stack element contains variable type and index
- types: local, interface, argument, temporary
Dependences at store instructions

store instructions check stack for copies
Stack depth at store instructions

- Depth 0: 98.4%
- Depth 1: 1.4%
- Depth 2: 0.0%
- Depth 3: 0.1%
Fixed register interface for basic blocks

- different register types (int, float, ...)
- consistent use of saved registers
Register allocation

• simple and fast algorithm
• javac compiler packs local variables
• fixed register interface for basic blocks
• argument registers are allocated in advance
• registers for local variables are allocated after stack variables
CACAOS's object and class representation
Just-in-time compilation

translation done during method invocation

static methods are called using address constants
Run time type checks

• relative tree numbering for classes

• interface hierarchy stored as pointer matrix

• run time type check done in four machine instructions
Performance comparison

load/compile/run times in milliseconds
Performance comparison

![Performance comparison diagram](image)

- espresso
- java_cup
- javac
- Toba
- JavaLex

performance relative to JDK-interpreter
Performance comparison

Performance relative to JDK-interpreter
Conclusion

main reasons for CACAO's efficiency

• efficient just-in-time compilation
• efficient use of machine registers
• efficient object and class representation
• efficient synchronization
• fast exception handling and run time type check

http://www.cacaojvm.org/