CACAO - A 64 bit JavaVM Just-in-Time Compiler

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

Abstract

This article describes the design and implementation of CACAO, a just in time compiler for Java. The CACAO system translates Java byte code on demand into native code for the ALPHA processor. During this translation process the stack oriented Java byte code is transformed into a register oriented intermediate code. Local variables and stack locations are replaced by pseudo registers eliminating the 32 bit restriction on address types. A fast register allocation algorithm is applied to map the pseudo registers to machine registers. During code generation, field offsets are computed for proper alignment on 64 bit architectures. Even though the CACAO system has to incur loading and compilation time, it executes Java programs up to 85 times faster than the JDK interpreter, up to 7 times faster than the kaffe JIT compiler. It is slightly slower than equivalent C programs compiled at the highest optimization level.

1 Introduction

Java's [AG96] success as a programming language results from its role as an Internet programming language. The basis for this success is the machine independent distribution format of programs with the Java virtual machine [LY96]. The standard interpretive implementation of the Java virtual machine makes execution of programs slow. This does not matter if small applications are executed in a browser, but becomes intolerable if big applications are executed. There are two solutions to solve this problem:

- specialized JavaVM processors,
- compilation of byte code to the native code of a standard processor.

SUN took both paths and is developing both Java processors and native code compilers. We chose to go for native code compilation since it is more portable and gives more opportunities for improving the execution speed. Compiling to native code can be done in two different ways: compilation of the complete program in advance or compilation on demand of only the functions which are executed (just in time compiler, JIT). Our JIT compiler is described in detail in [Gra97] (in German) and is freely available via the world wide web.

1.1 Previous Work

The idea of machine independent program representations is quite old and goes back to the year 1960 [Ste61]. An intermediate language UNCOL (UNiversal Computer Oriented Language) was proposed for use in compilers to reduce the development effort of compiling many different languages to many different architectures. The design of the JavaVM has been strongly influenced by P code, the abstract machine used by many Pascal implementations [PD82]. P code is well known from its use in the UCSD Pascal system. There have even been efforts to develop microprocessors which execute P code directly.

The Amsterdam compiler kit [TvSKS83] [TKLJ89] uses a stack oriented intermediate language. This language has been designed for fast compilers which emit efficient code. The intermediate representation of the Gardens Point compiler project is also based on a stack machine called *Dcode* [Gou97]. *Dcode* was influenced by Pascal P code. Both *Dcode* interpreters and code generators for different architectures exist.

The problems of compiling a stack oriented abstract machine code to native code are well known from the programming language Forth. In [Ert92] and his thesis [Ert96] Ertl describes RAFTS, a Forth system that generates native code at run time. Translating the stack operations to native code is done by translating the operations back to expressions represented as directed acyclic graphs as an intermediate step. In [EM95] he translates Forth to native code using C as an intermediate language. In this system the stack slots are translated to local variables of a function. Optimization and code generation are performed by the C compiler.

In his thesis [Fra94], Franz claims that generating native code on the fly at load time is faster than loading a much bigger native code image from a floppy or hard disk. Franz uses a compressed representation of the abstract syntax tree as an intermediate representation. Generating native code is so fast that this idea has been extended to dynamic run time reoptimization [Kis97]. Specializing and optimizing code at run time is also performed in the Self system [US87] and in some Prolog systems [KB95].

The first implementations of JIT compilers became available last year for the browsers from Netscape and Microsoft on PCs. They were followed by Symantec's development environment. Silicon Graphics developed a JIT compiler for the MIPS processor.

A public domain JIT compiler for several architectures is the kaffe system developed by Tim Wilkinson (`http://www.kaffe.org/`). For all the above mentioned systems, no publicly available description of the compilation techniques exists.

The SUN JIT compiler suite [CFM+97] is available for Intel and Sparc processors. The JIT compiler uses the same implementation of the virtual machine as the interpreter and its performance suffers because of the compatibility restrictions.

The translation scheme of the *Caffeine* system is described in [HGH96]. It supports both a simple translation scheme which emulates the stack architecture and a more sophisticated one which eliminates the stack completely and uses registers instead. *Caffeine* is not intended as a JIT compiler. It compiles a complete program in advance. The same group also developed an optimizing JavaVM compiler called NET (native executable translation) which is described in a recent article [HCJ+97].

Harrisa [MMBC97] is a Java system which contains an interpreter for dynamically loaded byte code and a compiler. The compiler translates a complete program and the libraries in advance to C code. A C compiler is then used to generate machine code.

DAISY (Dynamically Architected Instruction Set from Yorktown) is a VLIW architecture developed at IBM for fast execution of x86, PowerPC, S/390 and JavaVM code. Compatibility with different old architectures is achieved by using a JIT compilation technique. The JIT compilation scheme for the JavaVM is described in [EAH97].

2 The Java Virtual Machine

The JavaVM is a typed stack architecture [LY96]. There are different instructions for integer, long integer, floating point and address types. Byte and character types have only special memory access instructions and are treated as integers for arithmetic operations. The main instruction set consists of arithmetic/logical and load/store/constant instructions. There are special instructions for array access and for accessing the fields of objects (memory access), for method invocation, function call and type checking. A JavaVM has to check the program for type correctness and executes only correct programs. The following examples show some important JavaVM instructions and how a Java program is represented by these instructions.

`iconst_x`	push integer constant with value x
`iload n`	load contents of local variable n
`istore n`	store stack top in local variable n
`iadd`	sum of two topmost stack elements
`imul`	product of 2 topmost stack elements

The Java assignment statement `a = b + c` is translated into

```
iload b  ; load contents of variable b
iload c  ; load contents of variable c
iadd     ; compute b + c
istore a ; store stack top in variable a
```

Figure 1 shows the contents of the stack before and after execution of each instruction. Prior to the first instruction, and after the last instruction, the stack is empty.

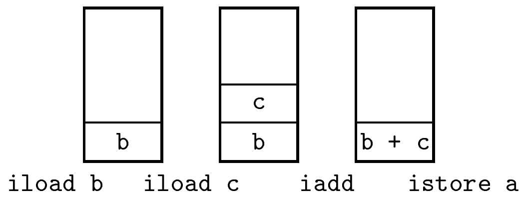

Figure 1: JavaVM stack operations

The `iload b` instruction pushes the contents of the local variable `b` onto the stack. `iload c` works in a similar way. `iadd` adds the two elements at the top of the stack, pops these two values and pushes the sum onto the stack. The `istore a` writes the value at the topmost stack position into the local variable `a` and pops this value.

Accessing the fields of objects is handled by the instructions `getfield` and `putfield`. `getfield` expects an object reference on the stack and has an index into the constant pool as an operand. The index into the constant pool must be a reference to a pair containing the class name and a field name. The types of the classes referenced by the constant pool index and by the object reference must be compatible, a fact which is usually checked statically at load time. The object reference has to be different from the `null` pointer, a fact which must usually be checked at run time.

Array access is handled by the `aload` and `astore` instructions. Separate versions of these instructions exist for each of the basic types (`byte`, `int`, `float`, `ref`, etc.). The `aload` instruction expects a reference to an array and an index (of type `int`) on the stack. The array reference must not be the `null` pointer. The index must be greater than or equal to zero and less than the array length.

The Java method invocation `o.print(a + 3);` is translated into

```
aload o  ; load object pointer o
iload a  ; load contents of variable a
iconst_3 ; push constant 3
iadd     ; compute a + 3
invokevirtual print ; call o.print
```

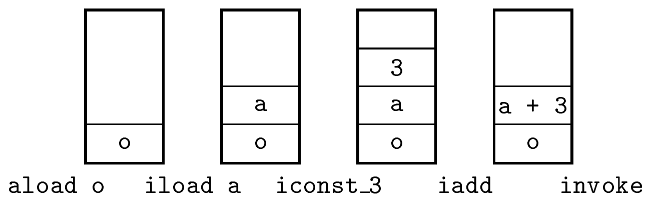

Figure 2: JavaVM method invocation

Figure 2 shows the contents of the stack for a method invocation. Each method has its own virtual stack and an area for local variables. At method invocation, the arguments are popped from the stack of the caller and copied into the first local variables of the called method. The stack of the callee is initially empty. After execution of a `return` instruction, the called method returns to its caller. If the called method is a function, it pops the return value from its own stack and pushes it onto the stack of the caller.

Only the behavior of the `invoke` and `return` instructions has been described. The concrete implementation is defined by the implementor of the abstract machine. One possible approach is to pass arguments via the stack and, instead of copying the arguments to the local variables, simply to adjust the stack pointer accordingly. Another solution is to pass the arguments via a register interface and use register windows or register coloring to achieve efficient parameter passing.

The `instanceof` and `checkcast` instructions are used for subtype testing. Both expect a reference to an object on the stack and have an index into the constant pool as operand. The index must reference a class, array or interface type. The two instructions differ in their result and in their behavior if the object reference is `null`.

The Java compiler computes the variable slots for local variables. Variables which are not simultaneously active are allowed to share the same slot. Variables of type `long` or `double` use two 32 bit sized slots. All other types including addresses use one slot.

3 Translation to Register Form

The architecture of a RISC processor is completely different from the stack architecture of the JavaVM. RISC processors have large sets of registers. (The Alpha has 32 integer registers and 32 floating point registers which are both 64 bits wide.) They execute arithmetic and logic operations only on values which are held in registers. Load and store instructions are provided to move data between memory and registers. Local variables of methods usually reside in registers and are saved in memory only during a method call or if there are too few registers.

If JavaVM code is translated to machine code, the stack is eliminated and temporary registers replace the stack slots.

3.1 Machine code translation examples

The example expression `a = b * c + d` has the JavaVM code

```
iload b   ; load contents of variable b
iload c   ; load contents of variable c
imul      ; compute b * c
iload d   ; load contents of variable d
iadd      ; compute (b * c) + d
istore a  ; store stack top in variable a
```

and will be translated to the following two Alpha instructions (the variables a, b, c and d reside in registers):

```
MULL b,c,tmp0  ; tmp0 = b * c
ADDL tmp0,d,a  ; a = tmp0 + d
```

The code generated for a method invocation is oriented to the calling conventions of the Alpha architecture for easy integration of native methods. The stack frame only contains spilled registers (including the return address register) and nothing else. The parameters are passed via argument registers.

The example method

```
public int add (int a, int b) {
    return a + b;
    }
```

has the JavaVM code

```
iload a ; load contents of variable a
iload b ; load contents of variable b
iadd    ; compute a + b
ireturn ; return method result to caller
```

and will be translated to the following two Alpha instructions (the parameters `a` and `b` reside in parameter registers `a1` and `a2`; this method is a leaf method and therefore does not need to save registers):

```
ADDL a,b,v0  ; resultreg = a + b
RET          ;
```

Invoking the method with the command `o.add(x * y, z - 3)` has the JavaVM code

```
aload o  ; load reference to o
iload x  ; load contents of variable x
iload y  ; load contents of variable y
imul     ; compute x * y
iload z  ; load contents of variable z
iconst_3 ; push constant 3
isub     ; compute z - 3
invokevirtual add ; call o.add
```

and will be translated to the following Alpha instructions (`mp` is a temporary register, `a0`, `a1` and `a2` are argument registers, the object pointer `o` is passed as the first argument):

```
MOV  o,a0      ; argreg0 = o
MULL x,y,a1    ; argreg1 = x * y
SUBL z,3,a2    ; argreg2 = z - 3
LDQ  mp,0(a0)  ; load class pointer into mp
LDQ  mp,add(mp); load address of add
JSR  (mp)      ; call o.add
```

The first three instructions set up the arguments, the last three instructions implement the method invocation (see figure 3 for a description of object and virtual function table layout).

3.2 Intermediate representation

The CACAO system does the translation to machine code in four steps. First, basic blocks are determined. Then, the JavaVM is translated into a register oriented intermediate representation, the registers are allocated, and finally machine code is generated. The intermediate representation is oriented towards a RISC architecture target and assumes that all operands are in registers (assuming an unlimited number of registers). In the following, all the intermediate code instructions are covered.

`LOADCONST` (type) #value `->` dest
: writes a constant value of the specified type (int, long, float, double, address) into the destination register.

`MOVE` (type) src `->` dest
: copies the source register into the destination register.

`OP1` (operator) src `->` dest
: executes an operation with one source and one destination register. The operator defines the kind and type of operation (e.g. `INEG`).

`OP2` (operator) src1, src2 `->` dest
: is similar to OP1, but with 2 source registers (e.g. `IADD`).

`OP3` (operator) src1, src2, src3 `->` source
: is similar to OP1, but with 3 source registers.

`MEM (LOAD)` (type) offset (src) `->` dest

`MEM (STORE)` (type) src2 `->` offset (src1)
: loads a value from memory into a register or stores a register into memory. The type specifies the type of the memory operation.

`BRA` (operator) address src1 src2 `->` dest
: is used for all kinds of branch and jump instructions, as specified by operator. The address references a target basic block.

`TABLEJUMP` (table) src
: is used for branching via a jump table.

`METHOD` (op) (descriptor) src1, ... `->` dest, exception
: is used for all kind of method invocations as specified by op (`INVOKESTATIC`, `INVOKESPECIAL`, `INVOKEVIRTUAL`, `INVOKEINTERFACE`). Parameters for the method are passed using an unlimited number of registers. The return value of the method is written into the destination register.

`DROP` register
: is a pseudo instruction telling the register allocator that the register is no longer used.

`ACTIVATE` register
: is the opposite instruction to `DROP`.

3.3 Translation scheme

The second pass of the compiler translates each JavaVM `load` or `store` instruction into a corresponding intermediate code `MOVE` instruction using a new register as the destination register in the case of a `load`. Always using a new register yields code in a similar form to static single assignment form [CFR+91], which is commonly used for compiler optimizations. A JavaVM `iadd` instruction is translated into a `OP2` instruction, again using a new destination register.

This naive translation scheme would generate many `MOVE` instructions. Therefore `MOVE` instructions are generated lazily. The translator keeps a table which tracks which registers should contain the same values. Instead of generating a `MOVE` instruction, the translator enters the register into the table. If the translator should later generate a `DROP` instruction, it deletes the register from the table. When the end of a basic block is reached, the corresponding `MOVE` instruction is generated for all registers remaining in the table. But for most basic blocks, the stack, and therefore the register table, is empty at the end or else the registers are compatible with the dependent basic blocks.

The example expression `a = b * c + d` with the JavaVM code

```
iload b   ; load contents of variable b
iload c   ; load contents of variable c
imul      ; compute b * c
iload d   ; load contents of variable d
iadd      ; compute (b * c) + d
istore a  ; store stack top in variable a
```

will be translated into the following intermediate code:

```
OP2(IMUL) b, c, t2 ; t0 == b, t1 == c
OP2(IADD) t2, d, a ; t3 == d, t4 == a
```

The `iload b` instruction does not generate a `MOVE` instruction for moving local variable `b` to stack location `t0`. It just marks in the register table that register `t0` and `b` are equal. Similarly the `iload c` instruction marks the equivalence of `t1` and `c`. In this example, all `load` and `store` instructions are eliminated. Registers `t0`, `t1`, `t2`, `t3` and `t4` may have existed in the register table for some time, but no `MOVE` instruction has been generated.

At a control flow join of two basic blocks where the stack is not empty and the stack slots are represented by different registers, a `MOVE` instruction is generated in one of the basic blocks. The register allocator tries to assign the same hardware register to the same stack slots so that the `MOVE` instruction can be eliminated.

3.4 Register allocation

Because a just in time compiler generates code at run time, it must be fast. Expensive register allocation algorithms, like graph coloring, cannot be used. We therefore designed a simple and fast scheme.

There are two different sets of registers: registers for stack slots and registers for local variables. First, registers for stack slots are assigned. Afterwards, the remaining registers are assigned to the local variables which are active in the whole method.

All registers are assigned to a CPU register at the beginning of a basic block. An existing allocation is left unchanged. The allocator scans the instructions and, for each instruction which activates a register and to which no CPU register has been assigned, a new CPU register is selected. If the allocator has run out of CPU registers, the register is spilled to memory.

There exist some conventions for the assignment of registers when calling methods. To prevent unnecessary copy instructions at a method call prior to the allocation pass, pseudo registers which are method parameters or return values are assigned the correct register (pre-coloring).

4 The Complete System

Generation of native code is only a small part of the complete CACAO system. The run time system is a major part. The run time system of the JDK is written mainly in Java and is distributed as class files. The Java methods call some native functions which we implemented for the ALPHA and which are contained in the CACAO system. The following subsections describe concepts of CACAO which are not directly related to code generation.

4.1 Object layout

The CACAO object and class descriptor layout has been designed for fast access and low memory consumption (see fig. 3). The SUN JDK represents an object by a cell with two pointers: the first points to the instance data of the object, the second to the class descriptor [HGH96]. Our representation eliminates one unnecessary indirection by having the object itself contain the pointer to the class descriptor and the instance

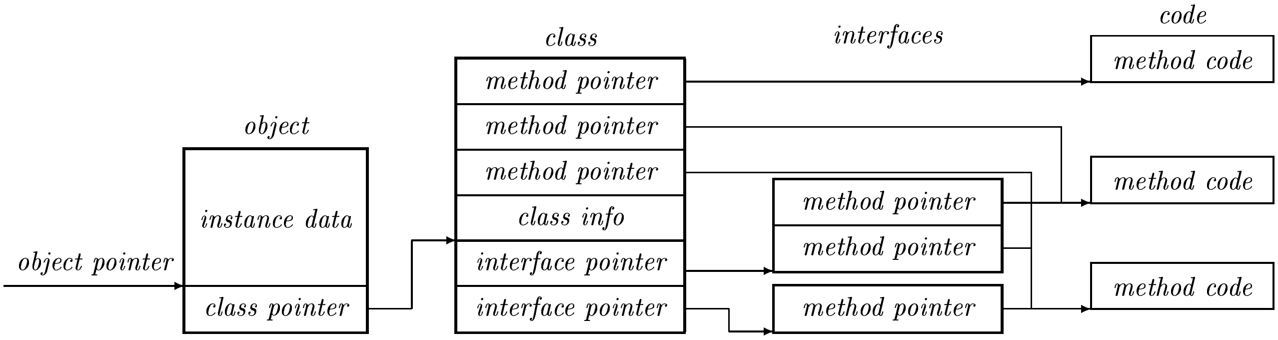

Figure 3: CACAO object and compact class descriptor layout

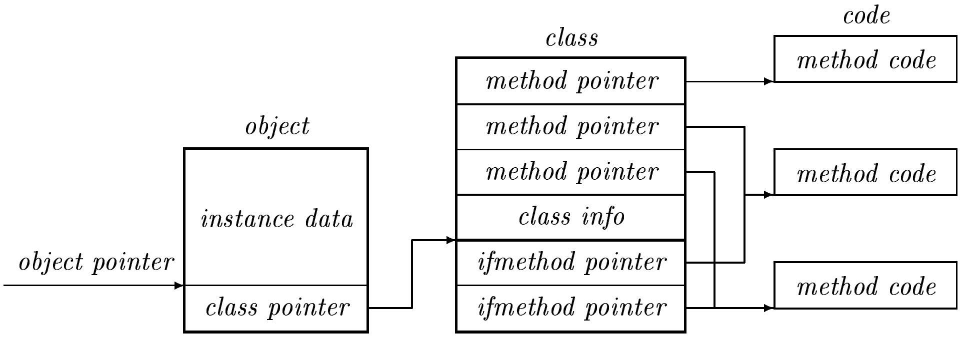

Figure 4: CACAO object and fast class descriptor layout

data. The Alpha architecture requires pointers to be 64 bits. Therefore field offsets in objects are computed to obtain correct 64 bit alignment of references, long integers and double floating point values. Since Alpha processors do not support 8 bit and 16 bit loads or stores, bytes and shorts are stored as 32 bit quantities aligned on 32 bit boundaries. Only byte, character and short arrays are stored in a compact representation since they can be large, and the saving in memory is worth the more expensive access.

In addition to other information, the class description contains the virtual function table. To call a method, two memory access instructions are necessary (load the class pointer, load the method pointer) followed by the call instruction. Java supports multiple subtyping via interfaces. Currently we are changing the representation for interfaces. The original representation (see fig. 3) needs one additional indirection, but usually consumes less space. In this compact layout scheme, the class table contains, at negative offsets, the interface table containing pointers to the interface virtual function tables.

In the faster scheme, we store interface methods in an additional table at negative offsets from the class pointer (see fig. 4). Segregating the interface virtual function table keeps the standard virtual function table small and allows interface methods to be called with just two memory accesses. The memory consumption of virtual function tables containing interface and class methods would be *number of (classes + interfaces) * number of distinct methods*. The memory consumption of the interface tables is only *number of classes which implement interfaces * number of interface methods*. We use coloring to reduce the number of distinct offsets for interface methods further. Compaction methods which reduce the size of the interface tables as described in [VH96] would increase the interface call overhead.

4.2 Method layout

The code of a method needs access to constants (mostly address constants). Since a global constant table would be too large for short addressing ranges and, because methods are compiled on demand, every method has its own constant area which is allocated directly in front of the start of the method code (see fig. 5). A register is reserved which contains the method pointer. The constants are addressed relative to the method pointer.

During a method call, the method pointer of the

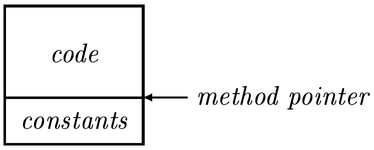

Figure 5: CACAO method layout

calling method is destroyed, but the return address is stored in a register which is preserved during execution of the called method and has to be used for returning from the method. After a method return, the method pointer of the calling method is recomputed using the return address. The following code for a method call demonstrates the method calling convention:

```
LDQ cp,(obj)      ; load class pointer
LDQ mp,met(cp)    ; load method pointer
JSR ra,(mp)       ; call method
LDA mp=ra+offset ; recompute method pointer
```

4.3 Just in time compilation

Machine code can be generated at either load time or at run time when a method is called. Compilation at load time simplifies the compiler and gives more opportunities for optimization. The drawback is that many methods are compiled which are never used. Therefore, CACAO translates methods just in time when they are called.

When class files are loaded, the virtual function tables and the interface tables are initialized with a pointer to a stub routine which invokes the compiler. This stub routine can determine its caller and invokes the compiler with the corresponding data. After the compiler has finished translating the method, it updates the method pointer in the virtual function table. The same method can be reached by different virtual function tables. The compiler only updates the table entry of the caller. But if the compiler is invoked, it determines whether code has already been generated. In that case, it only updates the pointer in the virtual function table. Thereafter, the newly generated code is executed.

For static functions the address of the function is stored in the constant area of a method. Neither the address of a virtual function, nor a static function are stored in the code area. This makes the updating of the address easy and prevents performance degradation of instruction caches.

4.4 Exception handling

The use of exception handling is quite common in Java. Typical exceptions are references to the `null` pointer, array index out of bounds or division by zero. To achieve portability across different architectures, checks are inserted at appropriate places. For example, before accessing a field of an object, the object reference is checked against zero. This is implemented by a single branch instruction which branches to the exception code. Because the branch is easy to predict, it executes very fast on modern processors. An array bound check is an unsigned comparison of the index against the array length. These checks are quite frequent, but can be eliminated in many cases. It is possible to move a loop invariant null pointer check before the loop or to eliminate a bound check.

Exception handlers are usually implemented by creating a linked list exception handling data structure when entering a `try` block and by discarding the structure when leaving the protected block. Since the use of exceptions is common in Java, we implemented a different scheme. Our exception system employs functions with two return values: one is the real result value, the second is the exception value. After each method call, the exception register is checked and, if it is non-zero, the exception handling code is executed. Since an exception is rarely raised, the branch is easy to predict and cheap. Entering and leaving a `try` block has no associated cost.

4.5 Run time type checking

A type inclusion test is a procedure to decide whether two types are related by a given subtype relationship. In Java, a run time type check results either from type casts or from explicit type checks (`instanceof`). In the JavaVM, the instructions `instanceof` and `checkcast` are used for subtype testing. Since Java currently does not support parametric polymorphism, type casts are used frequently. Therefore, the implementation of type checks has some affect on the performance of Java programs.

In [VHK97], we describe four different fast constant time type check methods. The fastest and most compact is not suited for the Alpha processor because it requires byte sized memory access. We therefore implemented a run time type check as a bit test in a bit matrix which contains the subtype relation.

4.6 Garbage Collection

Since Java relies on automatic storage reclamation, we also had to implement a garbage collector. To make

	JavaLex	javac	espresso	Toba	java_cup
run time on SparcStation 20 (in seconds)					
JDK	176.3	45.0	24.9	59.8	8.3
GUAVA	80.8	17.8	-	36.8	6.2
TOBA	38.8	12.8	4.9	20.1	2.9
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
CACAO total	2.65	4.74	3.17	4.58	1.52
load	0.18	0.61	0.38	0.18	0.21
compile	0.39	1.21	0.91	0.55	0.81
run	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/GUAVA	2.18	2.53	-	1.62	1.34
speedup JDK/TOBA	4.54	3.51	5.08	2.98	2.86
speedup JDK/kaffe	3.01	1.04	0.7	-	1.17
speedup JDK/CACAO	11.24	3.90	2.74	7.01	2.30

Table 1: comparison between JDK, GUAVA, TOBA and CACAO

it robust and easy to implement we kept the interface to the rest of the system as small as possible. There are functions to get new memory from the heap and to announce global variables that could contain references to the heap. As soon as garbage collection seems necessary – that means when too much heap memory has been consumed by allocation – a conservative mark-and-sweep algorithm is applied to release objects that are no longer needed [JL96].

For this purpose the garbage collector marks – starting from the announced global variables – all objects in the heap that could be found by following any pointer in any already marked object. Everything that does not get marked in the process will then be released. To distinguish pointers from integer data inside the objects, a simple (conservative) assumption is made: Anything that points at a valid object in the heap is treated as pointer and prevents the destruction of this object. Luckily, the address space of the ALPHA is 64 bit wide and so a coincidential match of an integer number with a valid pointer occurs rarely enough to make the collector work well.

5 Results

To evaluate the performance of CACAO we compared it with Sun's JDK and with kaffe version 0.8 (see section 1.1). In the last minute we got access to the JIT compiler from Digital (version 1.1.1 beta). We assume that the beta release has some problems because in one example it is 3 times slower than the JDK interpreter. Therefore we included only results where we believe they are correct. We also compared CACAO with two other JavaVM to native code compilers for the SPARC processor:

- Guava is a just in time compiler like CACAO which translates class files at run time into machine code
- Toba is a system which translates JavaVM instructions into C code. A standard C compiler is used to generate machine code. The measured time for Toba does not include compilation time, giving better results for Toba.

Neither system is available for the ALPHA, whereas CACAO currently supports only the ALPHA. We therefore compared all systems against the JDK interpreter, assuming that a SPARC processor and an ALPHA processor are similar and that the implementation of the JDK interpreter is similar.

The benchmark programs and the run time data for Guava and Toba were taken from the Toba homepage. We tested CACAO with exactly the same programs and the same data. `JavaLex` is a scanner generator, `javac` is the Java compiler from the JDK compiling the Toba sources, `espresso` is another Java compiler, `Toba` is a JavaVM to C compiler and `java_cup` is a parser generator.

	sieve	addition	linpack
run time on 21064A 300MHz (in seconds)			
JDK	83.2	138.76	1.6
kaffe	9.14	12.2	0.34
Digital JIT	7.27	5.33	-
GCC -O3	2.0	1.40	-
CACAO total	4.57	1.69	0.81
run	4.31	1.42	0.58
CACAO -cbnf total	3.46	1.62	0.33
run	3.31	1.42	0.13
relation of run time			
speedup JDK/kaffe	9.10	11.8	4.7
speedup JDK/Digital JIT	11.4	26.0	-
speedup JDK/CACAO	18.2	82.1	2.0
speedup JDK/CACAO -cbnf	24.1	85.7	4.8
speedup JDK/GCC	41.6	99.1	-
CACAO -cbnf run/GCC	1.66	1.01	-

Table 2: comparison between JDK, CACAO and GCC

Table 1 gives the run times. For CACAO, it also shows the load time and the compile time for each benchmark on an ALPHA workstation with a 300MHz 21064a processor. The CACAO system is between 2 and 7 times faster than the kaffe system. In nearly all cases, it is faster than Guava and Toba. Only when the compile time is high the Toba system is faster.

Table 2 compares the CACAO system with the Digital JIT compiler and a C compiler. `sieve` is the well known prime number computation program, `addition` is a loop with a simple addition and `linpack` is a floating point intensive program. The option `-cbnf` of CACAO disables array bound checks and precise floating point exceptions. Since C does not do implement these checks, it is fairer to make the comparisons with checking disabled. The CACAO system is only a factor of 1.66 slower than C and up to 7 times faster than kaffe.

6 Conclusion and further work

We presented an efficient layout for objects and classes in Java, a technique for translating the JavaVM to efficient native code for RISC processors and a novel implementation of exceptions. The CACAO system uses these techniques and executes Java programs up to 85 times faster than the JDK interpreter. It is only 1.01 to 1.66 times slower than an equivalent C program compiled with maximum optimization. CACAO can be obtained via the world wide web at `http://www.complang.tuwien.ac.at/java/cacao/`.

We plan to add instruction scheduling to the code generator assuming that this will help close the gap in speed with C. We will also split up code generation for method invocation to give the instruction scheduler more possibilities for moving instructions. We will integrate data flow analysis with safe bound check removal. Furthermore we will reduce compilation time by using state machines for code generation. Code generators for the MIPS architecture and the PowerPC are being developed.

Acknowledgement

We express our thanks to Anton Ertl, Thomas Gschwind, Nigel Horspool and Franz Puntigam for their comments on earlier drafts of this paper. We would also like to thank the reviewers for their helpful suggestions.

References

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

[CFM+97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3):36–43, June 1997.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control flow graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[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.

[Ert92] M. Anton Ertl. A new approach to Forth native code generation. In *EuroForth '92*, 1992.

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

[Fra94] Michael Franz. *Code Generation On the Fly: A Key for Portable Software*. PhD thesis, ETH Zürich, 1994.

[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.

[Gra97] Reinhard Grafl. CACAO: Ein 64Bit JavaVM Just-in-Time Compiler. Master's thesis, Technische Universität Wien, January 1997.

[HCJ+97] Cheng-Hsueh A. Hsieh, Maria T. Conte, Teresa L. Johnson, John C. Gyllenhaal, and Wen-mei W. Hwu. Optimizing net compilers for improved Java performance. *IEEE Computer*, 30(6):67–75, June 1997.

[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*, 1996.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection*. John Wiley & Sons, 1996.

[KB95] Andreas Krall and Thomas Berger. Incremental global compilation of Prolog with the Vienna Abstract Machine. In Leon Sterling, editor, *Twelfth International Conference on Logic Programming*, pages 333–347, Tokyo, 1995. MIT Press.

[Kis97] Thomas Kistler. Dynamic runtime optimization. In Hanspeter Mössenböck, editor, *JMLC'97 - Joint Modular Languages Conference*, Linz, 1997. LNCS 1204.

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

[MMBC97] Gilles Muller, Barbara Moura, Fabrice Bellard, and Charles Consel. Harrisa: a flexible and efficient Java environment mixing bytecode and compiled code. In *COOTS'97 - third Conference on Object-Oriented Technologies and Systems*, 1997.

[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.

[US87] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *OOPSLA '87 Proceedings*, pages 227 – 242, 1987.

[VH96] Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *6th International Conference CC '96*, pages 307 – 325, 1996.

[VHK97] Jan Vitek, Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In Toby Bloom, editor, *Conference on Object Oriented Programming Systems, Languages & Applications (OOPSLA '97)*, Atlanta, October 1997. ACM.